

POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



SNAKE: a privacy-aware online social network
providing anonymity of outsourced data at rest

Relatore: Prof. Gerardo PELOSI
Correlatore: Ing. Alessandro BARENGHI

Tesi di laurea di:
Michele BERETTA Matr. 782936
Alessandro DI FEDERICO Matr. 780708

Anno Accademico 2012–2013

To those who made this possible

Acknowledgments

First of all, we would like to thank our advisor, professor Gerardo Pelosi, who gave us the possibility to work on this fascinating project, following and giving us precious hints.

A special thanks goes to our co-advisor, Alessandro Barengi, for his remarkable competence, endless passion and his help, given so generously.

Finally, we thank the POuL association, there we met many people with whom we shared a lot.

I, Michele Beretta, wish to thank my mother, my father and my sister for their everlasting love and support. Another thanks goes to friends with whom I shared most of these five years at polimi, I am truly grateful for the beautiful moments we spent together.

I, Alessandro Di Federico, wish to thank Chiara, for the tenderness and the support she gave me during all these years. Ti amo. I also wish to thank my family, and in particular my mother: it is thanks to her if I am here. A special thank goes also to my good friends, such as Fabio and Lorenzo, for their deep friendship which emerges in unexpected moments. Finally, I would like to give special thanks all the hackers around the world, and in particular at POuL, who lead me to realize this project, in particular Radu, Daniele, Stefano and Michele.

Abstract

The usage of Online Social Networks (OSNs) is increasing day-to-day, reaching a user base unparalleled by previous online communication systems. However, the impressive amount of sensitive information stored on them introduces serious risks for the privacy of their users.

In this work, we aim to design an end-to-end encrypted OSN which allows easy one-to-one and many-to-many communication. The system should not require any knowledge about the underlying cryptosystem and scale in the number of users. The OSN client should also protect the metadata of communication, such as the social graph.

To achieve these objectives, we designed SNAKE, an HTML5 in-browser application interacting with a dumb storage server. We use the state of the art protocol for authenticated key agreement, FHEMQV, and provide two ways to authenticate public keys in-band: the Socialist Millionaire Protocol and a form of Web of Trust. Many-to-many communication is designed to handle dynamic groups in a scalable way. Thanks to a key graph system we are able to maintain a logarithmic cost to update the group key.

We also define policies for an honest storage server to protect communication metadata. In particular, we identify what are the essential information that must be stored in an intelligible form, for the storage server to be able to offer the service without major performance drops. With this approach we are able to leave almost no useful information for an attacker when the data is at rest. Finally, we evaluate the user experience, the system performance and the overall overhead for the end user.

Sommario

Gli OSN stanno acquistando una sempre maggiore popolarità. Tuttavia, l'enorme quantità di informazioni sensibili salvata su di essi introduce seri problemi di privacy per l'utente finale.

In questa tesi, il nostro obiettivo è progettare un OSN in cui i dati sono crittografati *end-to-end* e che permetta una semplice comunicazione tra singoli utenti e in gruppo. Il sistema non deve richiedere alcuna conoscenza a riguardo il sistema crittografico sottostante e deve scalare in presenza di un elevato numero di utenti. Il sistema deve anche occuparsi di proteggere le meta-informazioni della comunicazione, e in particolare il grafo delle relazioni tra gli utenti.

Per raggiungere questi obiettivi, abbiamo progettato SNAKE, un'applicazione HTML5 *in-browser* che interagisce con un *dumb* server. Per lo scambio chiavi abbiamo utilizzato lo stato dell'arte, l'algoritmo FHEMQV. Per l'autenticazione delle chiavi pubbliche abbiamo introdotto due opzioni: il protocollo del socialista milionario e una forma di *Web of Trust*. Lo scambio di messaggi all'interno di un gruppo è stata progettata per poter gestire gruppi dinamici in maniera scalabile. Infatti, grazie ad un sistema basato su un *key graph*, siamo in grado di mantenere un costo logaritmico per l'aggiornamento della chiave di gruppo.

Abbiamo definito una serie di politiche per un server onesto che intenda proteggere le meta-informazioni delle comunicazioni. In particolare, abbiamo identificato quali sono le informazioni essenziali da immagazzinare in maniera comprensibile dal server per far sì che il servizio funzioni senza eccessive perdite di prestazioni. Con questo approccio siamo stati in grado di non lasciare quasi alcuna informazione utile ad un attaccante che abbia accesso ai dati in condizione di riposo. Infine, abbiamo valutato l'esperienza utente, le prestazioni del sistema e il ritardo aggiuntivo introdotto per l'utilizzatore finale.

Contents

Introduction	1
1 State of the Art	5
1.1 Popular end-to-end encryption protocols	5
1.1.1 Pretty Good Privacy	5
1.1.2 Off-The-Record Messaging	7
1.2 Online Social Networks	9
1.2.1 «Overlay» OSN	10
1.2.2 Standalone	12
1.2.3 Peer-to-peer	14
1.3 Consideration on the state of the art	16
2 Proposed System architecture	19
2.1 Use cases	20
2.2 Analyzed scenarios	21
2.2.1 MALICIOUS-SERVER	22
2.2.2 HONEST-SERVER	23
2.3 Application distribution server	23
2.4 Data storage server	24
2.5 Client	26
2.5.1 Client architecture	26
2.5.2 Class diagram for models	29
2.5.3 Cryptographic primitives	31
2.5.4 Object serialization process	33
3 Friendship establishment	37
3.1 Notation	37

CONTENTS

3.2	MQV and its variants	38
3.2.1	Security features	40
3.3	Our protocol	42
3.3.1	Variations on FHMV-C	43
3.3.2	Public key authentication with SMP	45
3.3.3	Composing the protocol	46
3.3.4	Final key derivation	48
3.4	Web of Trust	48
3.4.1	Querying the WoT	49
3.4.2	Web of Trust table	51
3.4.3	Trust level computation	52
3.4.3.1	Trust computation in GnuPG	52
3.4.3.2	Trust computation in SNAKE	53
4	Group management	59
4.1	State of the art	59
4.1.1	Logical Key Hierarchy	60
4.1.1.1	Notation	60
4.1.1.2	Key graph	61
4.1.1.3	Insertion	62
4.1.1.4	Removal	63
4.1.1.5	Rekeying costs	65
4.1.2	Improvements over LKH	65
4.1.3	NSBHO	67
4.2	Proposed approach	68
4.2.1	Key graph approach	69
4.2.2	Join or leave a group and administrative privileges	69
4.3	Implementation details	70
4.3.1	Tree serialization and deserialization	70
4.3.2	Rekeying messages	71
4.4	Cost-benefit analysis and engineering tradeoffs	71
5	Anonymity of the stored data	77
5.1	Choice of the scenario	77
5.2	Preserving social graph secrecy	78

6	User experience	83
6.1	Design choices	83
6.1.1	Look and feel	83
6.1.2	Transparent signing and encryption	84
6.1.3	Multiple profiles	84
6.1.4	Group management	85
6.1.5	Friendship	86
7	Experimental Evaluation	89
7.1	WebCrypto API implementations	89
7.1.1	Key generation and key derivation	90
7.1.2	Encryption and decryption	91
7.1.3	Hashing and HMAC	93
7.1.4	Signing and verification	93
7.1.5	Password based key derivation	94
7.1.6	Final considerations	95
7.2	Use cases	96
7.2.1	Registration	97
7.2.2	Login	99
7.2.3	Sending Messages	100
7.2.4	Establishment of a friendship	101
7.2.5	Revocation of a friendship	102
7.2.6	Final considerations	103
8	Future developments	105
8.1	System architecture	105
8.2	Storage server	106
8.3	Messages	106
8.4	Groups	107
	Conclusions	111
	Acronyms	115
	Glossary	122
	Bibliography	123

CONTENTS

A Schemes	133
A.1 Database	133
A.2 System architecture	134

List of Figures

1.1	Example of a messaging session offering PFS	8
2.1	Scheme of system architecture	20
2.2	Scheme of the packing of a message	34
3.1	Exchanged messages in the FHMV-C protocol	39
3.2	Messages exchanged in the final friendship protocol	46
3.3	WoT examples	53
4.1	LKH: key graph representing a group of users	61
4.2	LKH: a new member joins the group	62
4.3	Node $[k_N]$ is splitted in nodes $[k_L]$ and $[k_R]$	62
4.4	LKH: a member is removed from the group	63
4.5	An element of $[k_L]$ node is redistributed in node $[k_R]$	64
4.6	Nodes $[k_L]$ and $[k_R]$ are merged into one node $[k_M]$	64
4.7	NSBHO: a new member joins the group	67
4.8	NSBHO: a new member joins a group whose associated tree is full	67
4.9	NSBHO real rekey cost for deletion	74
4.10	NSBHO number of keys inside the tree	74
4.11	NSBHO degrees removals statistics	76
5.1	Friendship handshake using a timestamp as primary key	79
6.1	Example of a forged message	84
6.2	List of user's profiles	85
6.3	SMP in action: authentication of Alice's public key	86
6.4	SMP in action: answer to Bob's question	87
7.1	Execution times for key generation and key derivation functions	90

LIST OF FIGURES

7.2	Execution times for encryption	92
7.3	Execution times for decryption	92
7.4	Execution times for hashing and HMAC	93
7.5	Execution times for signing and verification	94
7.6	Execution times for password based key derivation	94
7.7	Use case of user registration	98
7.8	Use case of login	99
7.9	Use case of sending messages	100
7.10	Use case of friendship establishment	102
7.11	Use case of friendship revocation	103
8.1	Subgroups	108
A.1	Scheme of the database tables	133
A.2	UML class diagram for models in SNAKE	134

List of Tables

1.1	Comparison of the state the art for privacy-aware OSNs	18
2.1	Summary of the models	29
2.2	Current status of WebCrypto API implementations	32
3.1	Comparison of the variants of the MQV algorithm	38
3.2	Log on the database after a friendship handshake	47
3.3	Summary of the different approaches to set up the WoT	49
3.4	List of friends of Alice's friends of the scenario in Figure 3.3b .	54
4.1	Rekeying costs for basic operations	65
4.2	Rekeying costs for the insertion of a new member	65
4.3	Comparison of rekeying cost between LKH and NSBHO	68
4.4	Results of regression to find NSBHO rekeying cost and tree size	75
7.1	NfWebCrypto and Polycrypt benchmark results summary . . .	95
7.2	Use cases benchmark results summary	104

LIST OF TABLES

List of Algorithms

3.1	Variation of the FHEMQV-C protocol	44
3.2	The Socialist Millionaire Protocol	57

LIST OF ALGORITHMS

Introduction

Online Social Networks (OSNs) such as Facebook [47] count hundreds of millions of users, which every day post personal information and share contents with friends, colleagues and family members. Due to their ease of use and pervasiveness, OSNs are also used to maintain business and commercial relations.

The enormous amount of sensitive information these platforms store is highly attractive for criminals, therefore adequate security measures are needed. So far, in most cases, OSN providers have had complete control over user data: the only restrictions came from the OSN's Terms of Service and the regulations of the various countries where the service was available.

The aim of this work is to allow OSNs' end users to have as much control as possible over the data they post and to design a service provider storing as little information as strictly necessary to offer the intended service.

For this reason, we present SNAKE¹, an OSN where most of the application logic resides in a client-side HTML5 application which can run completely in a web browser. All the user data is stored in encrypted form on a *dumb* storage server. Our system is user friendly and does not require any knowledge about the underlying cryptosystem. Thanks to a form of Web of Trust (WoT) and the Socialist Millionaire Protocol, we do not require out-of-band communication for public key authentication. In SNAKE, group-wide communication is handled through a *key graph* mechanism which allows to manage updates of the group key efficiently. Moreover, while the *dumb* storage server is outside the control of the end user, we include it in SNAKE's design assuming an honest provider. In this scenario we are able to preserve the privacy of the communication metadata, such as sender and recipient of a message, and more generally the

¹We took inspiration for the name from the Metal Gear Solid, a tactical espionage video game whose primary protagonist is called *Solid Snake*.

privacy of the social graph.

We benchmark the cryptographic primitives we use and the final overhead in terms of system responsivity for the end user in comparison with a similar system not using cryptography. Finally we explore future directions of research and development of SNAKE, in particular as a platform for applications not strictly related to an OSNs, but requiring a secure one-to-one and many-to-many communication channel.

In the vast world of OSNs, we took inspiration from Facebook in terms of features and target audience. In fact we target users willing to communicate privately with friends and family members, or more in general, people they know personally. For this reason we support two types of communication settings: one-to-one private messaging between users and many-to-many communication in a group. This is a quite obvious choice, since in other OSNs such as Twitter [52], posts are by default completely public and the user has therefore low privacy expectations. Since existing friends can be used to verify the identity (i.e. the public key) of new friends, trust in friends is a backbone of the whole system infrastructure. Therefore, we do not consider of primary importance scenarios where the two ends of the communication do not know each other, as it might happen for instance in a platform to support communication between whistleblowers and journalists.

For what concerns the system architecture, we proceed on the path of Friendegrity [33]. In particular we start from the issues the authors left open (see [32]): how to authenticate public keys without out-of-band communication, deploy the application to the client securely and do encryption efficiently on the client side. For public key authentication, we use the Socialist Millionaire Problem (SMP) described in [14], which verifies if two users know a common secret without disclosing any information about the answers given by the two ends of the communication, save whether the two answers match or not. We do not require the presence of an explicit pre-shared secret: a question is used to recall an implicitly shared secret. The idea of using such an approach comes from an improved version of the Off-the-Record Messaging (OTR) protocol presented in [2].

As an alternative way to authenticate public keys we also introduce a concept similar to WoT of Pretty Good Privacy (PGP), which in practice consists in examining friends of friends lists to collect enough confirmations that the public key to verify is authentic. The rest of the system mainly uses AES

for symmetric cryptography and Elliptic Curve Cryptography (ECC) when asymmetric cryptography is needed, in particular in the authenticated key agreement protocol, FHEMQV-C [91].

The outcome of our work is a typical client-server architecture where the two main actors are the HTML5 client and the storage server. All the components of the system have to be Free Open Source Software (FOSS). The HTML5 client is easily portable to different platforms and environments. In particular, we can deploy it in the form of a mobile application, a standalone desktop application or a web browser plugin. However, we foresee that most people will want to use the application directly in the web browser as a standard web page. For this reason, in our design we include a second, optional, server whose role is to distribute the client application to the end user. While its role is extremely simple, as it could even be hosted on a file server, it is a critical component that has to be managed by a trusted third party, since it has full control on the code users run. To mitigate this risk we briefly describe a distribution platform similar to the one of most GNU/Linux distributions.

The client itself is composed by HTML markup and JavaScript code, however, thanks to WebCrypto API [24], we use in almost all cases the cryptographic primitives provided by the web browser, which offer quasi-native performances and a sound implementation. To achieve high levels of usability, we transparently encrypt and sign outgoing data and verify integrity and decrypt incoming data, hiding all the cryptographic details from the end-user.

For what concerns the storage server, we model it as *dumb* entity whose role consists in performing Create, Read, Update and Delete (CRUD) operations. Note that the server uses a stateless protocol which handles all the request without enforcing any access control, except for the fact that only the original creator of a record can update or delete it.

We consider the storage server in two different scenarios: MALICIOUS- and HONEST-SERVER. In the former scenario, the storage server is seen as an adversary, and our aim is to protect the content of messages, profiles and so on from it. In the latter scenario, we assume the server is willing to collaborate with us to offer a service providing guarantees on the anonymity of the data at rest, that is in case an attacker gains access to a database dump.

In the future we intend to improve some aspects on the scalability of group management and of the storage server. In particular for the latter, we want to add support for a *federated* configuration, where the storage server hosts

user data but it also works as a gateway to other independent storage servers hosting different data.

An interesting development of SNAKE is adding support for push notifications, which would allow to implement a real-time chat application, possibly with an *OTR-like mode*, offering the guarantees described in its design paper [13]. SNAKE is also designed to easily integrate third party, untrusted and cryptography unaware applications. This plugin system is particularly suitable for applications such as an online collaborative office suite or a file sharing application.

The rest of this work is organized as follows. In Chapter 1 we illustrate the current state of the art and analyze aspects that need more attention. In Chapter 2 we provide details about the structure, the design and the technological aspects of SNAKE. In Chapter 3 we illustrate how a secure friendship relation is established. In Chapter 4 we describe how we handle in a scalable way group communication and in particular member revocation. In Chapter 5 we detail the guarantees about the anonymity of the social graph on the data at rest. In Chapter 6 we discuss the design choices for the front end of SNAKE. In Chapter 7 we show the performance of cryptographic primitives and the overhead for the end user introduced by the pervasive use of cryptography. In Chapter 8 we explore future directions for our work.

Chapter 1

State of the Art

In this chapter we analyze the state of the art from two points of view: we present some of the most popular solutions for end-to-end secure communications and we describe the current status of the research in the field of secure and privacy-aware OSNs. Finally, we provide some considerations on aspects which require particular attention such as scalability, usability and code origin.

1.1 Popular end-to-end encryption protocols

During the design of SNAKE we kept in high consideration some currently relatively popular and widespread solutions for secure communication not relying on a trusted third party, as it happens with the classical PKI scheme. In particular we took inspiration from PGP and OTR cryptographic schemes.

1.1.1 Pretty Good Privacy

PGP is a widespread tool to encrypt and sign arbitrary data. It was invented by Phil Zimmerman in 1991, as explained and detailed in an informal communication in [107], as a simple tool that would allow the average user to achieve strong privacy over his communications. After a troubled release history, PGP has been finally standardized in RFC 4880 [17]. Its main uses are digital signing and encryption of e-mail communications, encryption of files and digital signing of software packages for secure distribution. Its simplest use is the symmetric encryption of a file with a key, derived from a password, which has to be exchanged with the intended recipient through a secure communication channel. Since an actual secure communication channel is usually

not present, PGP employs PKC. Each user has one or more keypairs, whose public part has to be spread as much as possible and it is often published on the user's personal website, in the e-mail signature or, most importantly, on dedicated servers called *key servers*.

When Alice wants to send an encrypted message to Bob, she first encrypts it with a random symmetric key, then retrieves Bob's public key from one of the above mentioned sources, and attaches to her message the symmetric key encrypted with Bob's public key. If she wants to make the same message available to Charlie, she has to encrypt the same symmetric key with Charlie's public key and attach it too. When the recipient reads the message, he first decrypts the symmetric key with its private key and, with the obtained key, he is able to decrypt the message itself.

If Alice wants to send Bob a digitally signed message, she uses her private key to sign the message and attaches the signature to the message. Anyone in posses of Alice's public key can verify the signature.

A key point for the described system to work properly is the authenticity of public keys. In fact, if an attacker, Mallory, is able to make Bob believe Alice's public key is a public key whose associated private key is under Mallory's control, he can send Bob a message pretending to be Alice. In the same way, if Mallory can intercept an encrypted message for Bob, he can use his own private key to obtain the symmetric key used to encrypt the message and read its content. Then, he can re-encrypt the key with Bob's real public key and forward him the message. This kind of attack, known as man-in-the-middle (MITM), is completely transparent to both Alice and Bob.

To overcome this issue PGP introduces the concept of WoT. Basically when the public key of a user gets published on a key server, it is possible for another user to publish a digital signature over that public key using his own private key. When a user signs a public key he is saying he verified that the public key corresponds to the associated user. We can represent this structure as a directed graph where the vertices are the public descriptors of users (typically made of an e-mail address and the public key) and the edges are the signatures. The result is the so called Web of Trust. The trust derives from the fact that the signing user has verified, either over a secure channel or in person (at the so called «key signing parties»), that the signed public key actually belongs to the declared user.

The Web of Trust is helpful in several situations to make public key authen-

tication easier. Suppose Alice wants to write to her new friend, Bob. Suppose also both Alice and Bob are friends with Charlie. If Alice has verified Charlie's public key in person, and Charlie did the same with Bob, Charlie can publish a signature over Bob's public key, which is enough to make Alice trust Bob's public key, since she can verify Charlie's signature with his public key.

Despite these useful features PGP has a series of design flaws. First of all, encrypting a message for multiple recipients it is not scalable, since it requires to attach a different version of the symmetric key per recipient along with the ciphertext, which becomes unpractical if a user wants to send the same message to a large number of people.

As highlighted by Mike Perry in [80], the WoT suffers from a series of problems too. First of all, the graph presenting the WoT is public and it is possible to infer relationships among users (a signature over a user's public key likely means the two people know each other). Moreover, highly trusted users represent single points of failure, as their private keys, if compromised, would allow an attacker to sign a fake key which would in turn be trusted by a large amount of users. This is due both to the transitivity of the trust in the PGP model and to the fact that the whole graph cannot be authenticated, which allows an attacker to mangle it, possibly hiding significant parts.

Besides the technical issues, and despite its relatively large adoption, PGP is considered to be still rather complicated for the average user, since it requires to have a basic understanding of how PKC works, and how to make it work properly. Big steps have been done to improve its usability, for instance with friendly GUIs but, as shown in a in-depth analysis of a PGP GUI [103], there are still serious usability issues.

1.1.2 Off-The-Record Messaging

OTR is a cryptographic protocol designed to agnostically work over any existing instant messaging protocol such as Skype, XMPP and the now defunct Windows Live Messenger. OTR was created by Borisov et al. [13] to overcome some problems present in protocols such as OpenPGP, in particular when compared to the privacy features of a real-life secret conversation. OTR aims to offer the following additional properties:

Perfect forward secrecy (PFS) The conversation secrecy must be guaranteed even if an eavesdropper records the encrypted communication be-

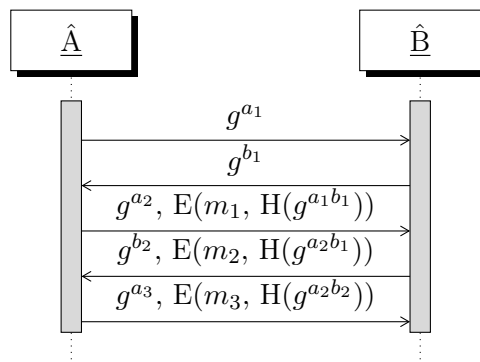


Figure 1.1: Example of a messaging session offering PFS. PFS is achieved sending a new public key at each message exchange and using a key derived from the latest available keypair.

tween Alice and Bob, and, in a later moment, recovers all their long-term secret key material. This properties is achieved using an ephemeral key for the encryption of the messages, derived through the Diffie-Hellman (D-H) key agreement protocol [29] and discarded as soon as the symmetric session key is obtained.

Figure 1.1 shows a short example session where PFS is achieved. We assume Alice and Bob agree on a prime number p , a primitive root g modulus p and choose two integers, respectively a_1 and b_1 , which we call *ephemeral private keys*. Also, let H be a cryptographically secure hash function. When the conversation begins, Alice and Bob compute over \mathbb{Z}_p and exchange a first pair of *ephemeral public keys* g^{a_1} and g^{b_1} . Then Alice sends a first message m_1 encrypted with $H(g^{a_1 b_1})$ along with a new public key g^{a_2} . At this point Alice can delete her ephemeral secret key a_1 . Bob decrypts m_1 with $H(g^{a_1 b_1})$, answers with a message encrypted with $H(g^{a_2 b_1})$ and g^{b_2} , and forgets b_1 . It is now information theoretically impossible for anyone, Alice and Bob included, to recover $g^{a_1 b_1}$, and thus (assuming the symmetric cipher is not broken) recover the plaintext of m_1 .

Repudiability Bob should be able to prove himself that a message he receives from Alice is actually authentic, but he shouldn't be able to prove that to a third party. This is the opposite of the well-known *non-repudiability* property that digital signature schemes, such as RSA, offer. To obtain this property, the exchanged messages are not digitally signed: integrity

is ensured computing a message authentication code (MAC) of the message using as key the hash of the key used for encryption: $H(H(g^{a_i b_j}))$. This means that both Alice and Bob can forge a message with an appropriate MAC and it is therefore impossible to prove to a third party who the author was.

Forgeability Short after Bob has verified that the received message was from Alice, the message must become forgeable by anyone. The OTR protocol ensure this property letting Alice reveal in clear the key used to compute the MAC over message m (i.e. $H(H(g^{a_i b_j}))$), right after she is sure Bob has used that value to verify m . In this way anyone is able to forge a message m' and compute a valid MAC for it.

We said digital signatures are not used due to the repudiability requirement, however there is a necessary exception to guarantee authentication: the first D-H exchange is digitally signed so that Alice is sure to be talking with Bob and viceversa.

The existence of long-term keypairs leads us back to the problem of public key authentication. While at first public keys were assumed to be verified out-of-band, in a later work [2], the authors of OTR introduced a new mechanism to authenticate public keys not requiring (explicit) out-of-band communication. In short, Alice asks Bob a question whose answer is known only to the two of them, and the answer is employed, together with the public key of both users, in the Socialist Millionaire Protocol, as described in [14]. SMP allows to determine whether Alice and Bob share a secret value, without revealing each other anything concerning the secret, except whether the two answers match or not. We will provide an in depth description of those mechanisms later on in Chapter 3.

1.2 Online Social Networks

Due to the high amount of attention received in the mainstream media, privacy in the OSN context has become a relevant research topic in the academic world too. The proposed solutions can be split in three categories with respect to how they store user data: distributed, centralized (or federated), and as an overlay to an existing OSN. In the following, we present these three

categories and focus on their advantages and problems paying particular attention to some aspects we consider relevant such as user revocation, group communication cost, public key authentication and location of the data and the application.

1.2.1 «Overlay» OSN

The idea of adding privacy to an existing OSN is quite attractive since no user migration is required, existing accounts and relationships are effortlessly preserved. A further advantage for the end user is the familiar user experience, since it is not necessary to deal with a new interface. On the other end the original service provider might not approve encrypted content on its platform, as it would prevent it from mining user profiles in order to offer highly targeted advertisement, which currently is one of the most widespread business models for OSNs. To overcome this issue Saikat et al. [44] proposed NOYB, a system to improve privacy of user profiles of an existing OSN in a way hard to detect for the OSN provider. To achieve this, NOYB does not directly encrypt profile fields but swaps them with those of another user of NOYB, using a pseudo-random substitution cipher. What gets encrypted is the index which maps a user to its profile fields. Suppose for instance Alice is female and 25 years old, her profile descriptor would be (Alice, F, 25). NOYB does not save that data on the underlying OSN, but splits the profile in two parts, (Alice, F) and (25), and substitutes them with data from other users of NOYB, for instance (Bob, M) from Bob and (32) from Charlie. A user in possess of Alice's key is able to decrypt the index used to associate these information correctly. This index is hidden into the profile picture through steganography techniques. This way, without the key, it is not even possible to tell whether NOYB is being used at all.

While being original and interesting, NOYB approach is viable only with small amount of data, namely profile details. Matthew et al. introduced fly-ByNight [65], which, being a Facebook application, still relies on a previous OSN but it is able to provide secrecy for one-to-one and one-to-many messages. The application encrypts messages on the client side in JavaScript and stores them on its own server. In case of multiple recipients, the message is re-encrypted by the flyByNight server through a proxy re-encryption scheme which basically shifts the computational cost of encryption (which is linear in

the size of the group) from the end user's client to an intermediate proxy.

flyByNight encryption is done on the client-side, and therefore can be easily tampered by the OSN to obtain plaintext or the secret key material itself. For this reason Luo et al. proposed FaceCloak [67], which is similar to flyByNight but relies on a browser extension instead of on a Facebook application, making it impossible for the OSN to alter it. It also strives to be undetectable by the OSN posting apparently sensible messages on Facebook, and storing the encrypted payload on a third party server. However, since fake messages are indistinguishable from unencrypted messages, each message has to be looked up on the third party server to check if an encrypted version is available.

Beato et al. [9] proposed Scramble, a completely client-side solution which does not involve any third party server but relies only on a browser plugin. The plugin works in an OSN-independent fashion, which makes it suitable even for blogs and other online systems. Messages are encrypted using the OpenPGP standard [17] to enforce Access Control Lists (ACLs). When the plugin detects a PGP message in the page, it is decrypted transparently on the fly.

One of the big disadvantages of the *overlay* approach is that it may lead to a Terms of Service violation, for instance on Facebook the user is not allowed to post fake profile information. Moreover, people not using the privacy enhancing application will see wrong and misleading information that strive to appear legitimate to the OSN or unintelligible encrypted data. For example ASCII-armored PGP message or series of *Chinese, Japanese, and Korean* (CJK) symbols as in [92].

A further problem of overlay based approaches is that, if no distinct OSN is created, most of the above systems rely, at least partially, on a third party server hosting metadata or the content itself. The actual advantage over standalone OSN is thus reduced to have at disposal existing relationships: a scalable and robust infrastructure is still required. In certain cases, such as [65], the external server might have a considerable amount of load not only from the network point of view, but also on the computational side.

As it has been pointed out in [99], adding a cryptography layer on a system which is mostly cleartext is dangerous, since it is always possible for an attacker to perform a downgrade attack, by making it impossible or very hard to post the encrypted message until the user gives up and sends it in clear anyways.

This kind of approach also suffers from the fact that the social graph and the messages' metadata are still available to the OSN, while a completely

distinct implementation might have a wider room for manoeuvre.

In conclusion, the problem with adding privacy as a layer over existing systems is general and hard to overcome, since strong user privacy is in perfect opposition with the business model of current OSNs, which obtain most of their revenue out of highly targeted advertisement. Widespread usage of these methods would inevitably lead to a strong effort by the service provider to detect encrypted messages and related plugins or applications and ban them. We thus deem this approach as not viable on the long-term.

1.2.2 Standalone

A first approach in building a standalone OSN consists in having a series of interacting federated servers. In the ideal situation, each user has its own individual server running the application. This way, the server can be almost completely trusted and privacy is preserved even if it stores unencrypted data. Giving access to the plaintext offers the benefits of a lower workload and the chance to perform range queries efficiently. An example of such an architecture has been proposed in Vis-à-Vis [93], where each member runs an Amazon EC2 instance.

While keeping the users' personal data on a private server offers a good privacy level, it is not very realistic to assume each member will be able to manage and afford a dedicated service. A relaxed version of the federation principle is at the base of another OSN which received a considerable amount of attention, Diaspora [35]. In Diaspora everyone can set up a *pod*, i.e. a node of the federated network, for both personal use and for other trusting people: the main difference if compared with Vis-à-vis is the fact that each pod can host account of multiple users. The ease of usage comes at the cost of trusting the owner of the pod, which may or may not be more trustworthy than current centralized OSN providers. A frequently proposed solution for this problem is to use the pod set up by a friend considered trustworthy. However, the trust in a friend is not perfectly stable and eternal, and in any case the user should always keep in mind the owner of the pod will always be able to read all his messages. In conclusion preserving the user's privacy in a federated environment could result to be even harder than dealing with a single entity, such as Facebook, since law enforcement against a single centralized provider would be easier. Moreover, storing unencrypted data exposes the OSN to

massive data leak in case of server compromise by an attacker or seizure by law enforcement.

Others have faced the challenge from another point of view, considering the storage provider, at least partially, an untrusted entity. De Cristofaro et al. worked on Hummingbird [28], a Twitter-like application which aims to preserve the privacy of a user willing to follow another user or a simple hashtag. To achieve this all the tweets and associated hashtags are encrypted by the client before being delivered to the remote server. Despite encryption, thanks to an Oblivious Pseudo Random Function (OPRF) mechanism based on Blind-RSA signatures, the service provider is still able to match, for each user feed, hashtags and their subscribers.

Tran et al. [97] proposed another system involving proxy re-encryption in which there is a private key x for each user i , which gets divided in two parts, x_{i_1} and x_{i_2} , where $x = x_{i_1} + x_{i_2}$. The former is given to the user, while the latter to the proxy. When a user i wants to send a message to a group, he encrypts it with x_{i_1} with El-Gamal, and stores it on the cloud. When j , who is a member of the group, wants to get access to the message, the proxy is responsible to convert the message to something j can decrypt. To achieve this, the proxy first encrypts with x_{i_2} the message and then decrypt it with x_{j_2} . Thanks to El-Gamal properties, this operation leads to the original message encrypted with x_{j_1} , which allows j to decrypt it. While this scheme does not allow the proxy to read the content of the message, it relies on an external trusted entity, a key manager, which has to generate and distribute the keypairs.

In Frientegrity, Feldman et al. [33], face the same problem but using a graph structure, introduced in [104], which performs key revocation in logarithmic complexity w.r.t. the number of members of the group. The other important focus in Frientegrity is the detection of *equivocation*, that is the situation in which different users have different, but internally coherent, views of the OSN. In their system, the storage provider can still offer different views to two users, however once it does, the users must be prevented from communicating between each other, otherwise the tampering would be detected. While we leave equivocation detection for future works, in Chapter 4 we further explore and improve key graph usage in the OSN group management context.

Du et al., in PrivOSN [31] ignore end-to-end encryption and focus their attention on minimizing the amount and size of trusted components in the system and strive to improve anonymity and privacy of relationships. In the

PrivOSN model the only component requiring full trust is the **pmonitor**, a tiny application running on the user’s machine whose role is to transparently encrypt/decrypt all the outgoing/incoming traffic generated by the real client. The real client, called **pclient**, runs in a sandboxed environment similar to Google NaCL [48], and cannot communicate with the network bypassing the **pmonitor**. The advantage of this decoupling relies in the fact that **pmonitor** should be open, small and simple, and therefore easy to analyze and review. On the anonymity side, PrivOSN introduces a component called **pproxy** which consists in a proxy server between the client and the server storing the encrypted data which forwards the traffic from and to the server and the client anonymously. This second level of decoupling makes it harder for the OSN service provider to monitor user interactions and thus relationships.

1.2.3 Peer-to-peer

Baden et al. proposed Persona [6], a decentralized platform offering a Facebook-like application, protecting the remotely stored data with a mix of Attribute Based Encryption (or ABE, as introduced in [90]) and classical PKC. Attribute Based Encryption offers high flexibility in ACL definition at the cost of an higher computational cost.

Jahid et al. in EASiER [53] continue on the path of Persona, proposing an efficient way to revoke access thanks to a minimally trusted third-party server. This external entity cannot decrypt the messages it receives but, through proxy re-encryption, manages revocation of users and attributes (i.e. group memberships) without forcing the original author to distribute a new key and to re-encrypt the data. This feature is particularly useful if compared with the linear cost in the size of the group that several previous works (such as [6, 28, 65]) require to ban a user. It is also worth noting that, unlike Tran et al. [97], proxy re-encryption does not require a third party key manager. On the other hand, proxy-encryption requires to rely on a third party server, which, although cannot collude with a banned user to decrypt messages sent after his removal, has to perform cryptographic operations for each request, which may be computationally intensive for large amounts of traffic.

In EASiER and Persona little attention was paid to the Peer-to-peer (P2P) network infrastructure. EASiER authors explored this aspect in DECENT [54]. They proposed a decentralized implementation of the illustrated approach

based on Distributed Hash Table (DHT), however performance tests lead to not completely satisfactory results. DECENT has been subsequently improved to reduce computational and network costs using social relationship as caches. This caching system, implemented in Cachet [77], consists in storing the updates of a user in his friends' nodes in unencrypted form. This way, friends work as a plaintext cache to speed up retrieval of most recent updates, while other updates have to be retrieved from the DHT and decrypted.

One of the first attempts to design a P2P OSN was made by Buchegger et al. with PeerSoN [16]. To ensure secrecy of the messages exchanged among users, PeerSoN makes use of symmetric and asymmetric encryption. As a first approach to the problem, and being more focused on other challenges such as data availability, the authors assume the existence of a PKI offering key revocation and quickly mention out-of-band authentication of friends' credentials. While this may sound unpractical, it fits well into one of the primary features declared in PeerSoN: offline direct data exchange through, for instance, mobile devices. The authors also highlight some of the big issue emerging in the design of a P2P OSN is data availability, how to exploit geographic diversity of the nodes to increase it and how to overcome the high resource demand in terms of storage for the end user.

Narendula et al., in porkut [76], present an approach similar to Diaspora but in a P2P setting. In fact, the user chooses a couple of friends who are responsible for hosting his data and enforcing the associated access policies. These nodes have to be chosen carefully and keeping in mind both their trust level and their capability to keep the user's data available. Liu et al. designed Confidant [62] in a similar way. The data is replicated in plaintext on a limited number of friend's machines. However they still require a hosted server to serialize the updates of the user, and assume that the keypairs involved in the system can be exchanged out-of-band securely. Moreover, they do not offer a way to evict a group member without recreating the group from scratch.

Graffi et al. in [43] design a P2P system enforcing ACLs through a naïve approach similar to PGP: content is encrypted with a symmetric key, which is attached to the message asymmetrically encrypted multiple times, once for each intended recipient. Public key authentication is bypassed using public keys themselves as identifiers.

Backes et al. [5] defined an generic API for a secure OSN platform which aims to enforce access control over user resources preserving the privacy of

social relations and anonymity of the user. In their work data is hosted unencrypted on a server operated directly by the data owner. While this approach has already been discussed, the authors introduce a new interesting point: ACLs are not kept in clear, but are composed by anonymous tokens associated with their permissions. These tokens can be either a pseudonym or a social relation (e.g. close friends). A user gets access to the resource revealing its pseudonym or proving to be in a certain relation through a zero-knowledge proof. The declared aim is to protect the social graph even in case of compromise of the storage server or seizure of data.

What makes the P2P configuration particular attractive, besides using an existing infrastructure without additional costs, is the fact that there is not a single (or a few) entities storing all the data, making thus such a network harder to monitor effectively. However, following the recent coming to light of nation-state scale wiretapping on large data backbones [57], such a large scale monitoring is no more implausible.

1.3 Consideration on the state of the art

The presented literature covers a large portion of the issues related to the design of an end-to-end secure and privacy-aware way to communicate. Table 1.1 on page 18 tries to summarize a subset of this aspects we consider of particular relevance.

Cost of user revocation Removal of a member from a group should scale well with the size of the group. This means it should not require to send a custom message to each member of the group (i.e. $O(n)$ cost). To achieve this, an external trusted entity is required (such as in EASiER and derivatives [53,54,77]) or some more sophisticated technique such as hierarchical key management illustrated in Frientegrity [33].

Public key authentication Public key authentication cannot be ignored and it should be easy for the average user. It should not require (explicit) out-of-band communication. Almost all of the analyzed works ignore this aspect, which we consider a key point to make cryptography usable without relying on a trusted third party. We consider of particular interest the method proposed for OTR in [2].

Code origin The choice of client origin is a compromise between ease of use and security. To offer an experience as close as possible to that of current OSNs, a secure OSN could be designed as a simple web application delivering an HTML page taking care of the encryption part and then storing the data remotely. The problem with this solution is the fact that the service provider can easily tamper with the code and compromise the keys. Moreover JavaScript cryptography is problematic from both a security and a performance point of view. For these reasons several of the mentioned works require the user to install a browser plugin. This way a native implementation of the cryptographic primitives can be used, and the client can be deployed through reliable channels. On the other side this plugins are frequently architecture- and browser-dependent. The third approach, typical of P2P systems, consists in installing a completely standalone application.

Data storage location The location of the data is a characteristic of high relevance, since it determines who retains control over data. The ideal solution would be the one where each user keeps his own data and directly distributes it to those interested. We deem unpractical to have a dedicated server per-user, so we mainly considered a P2P configuration. Distributed storage requires nodes to host data of other users to keep it available while the respective owners are offline. However this approach incurs in non-negligible network and storage overheads, as shown in [16]. While less than ideal, storing data on a dedicated server or on a federated network of public, non-personal servers is more realistic and can offer unmatched levels of data availability.

Cost of multiple-recipient messages Group messaging is a vital feature of an OSN, for instance to implement the *wall* application. For this reason we deem fundamental to have a constant cost in the number of recipients in terms of computation and message size. In particular we do not want to attach the key encrypted in as many different ways as the number of recipients ($O(n)$ cost). An option for this is to use proxy re-encryption to transform the sender's ciphertext in a ciphertext which can be decrypted by the recipient. While the usage of a proxy saves some computational effort due to the fact that, if a message is never requested by a specific recipient, no computation takes place, it simply shifts the cost from the

client to the proxy. If an appropriate revocation mechanism is in place, we deem the usage of a shared symmetric key to be the best option.

We also note that the all the proposed solutions which work as an overlay over an existing communication system, be it an OSN, the e-mail service (for PGP) or an instant messaging protocol (for OTR), cannot protect all the metadata of the underlying layer, which usually includes sender, recipient and other sensitive information.

Table 1.1: Summary of relevant aspects of the state of the art for privacy-aware OSNs. The *Revocation* column indicates the asymptotic amount of data to send to remove a user from a group, in the the group size n . *Public key authentication* indicates how public key authentication has been handled. *Code origin* indicates where the application runs and what is its origin. *Data location* indicates where the user data is stored. *Group message* indicates the asymptotic computational cost, in the group size n , required to send a message to a group, and which component is in charge of doing so. *Not considered (n.c.)* means that the specific aspect has been ignored or has been considered as a future development.

	Revocation	Public key authentication	Code origin	Data location	Group message	
					Cost	Where
NOYB [44]	$O(n)$	out-of-band	Firefox plugin	underlying OSN	$O(1)$	client
flyByNight [65]	n.c.	out-of-band	Facebook app.	dedicated server	$O(n)$	proxy
FaceCloak [67]	n.c.	out-of-band	Firefox plugin	dedicated server	$O(1)$	client
Scramble [9]	$O(n)$	out-of-band	Firefox plugin	underlying OSN	$O(n)$	client
Vis-à-Vis [93]	N/A	out-of-band	web app.	own server	N/A	N/A
Diaspora [35]	N/A	N/A	web app.	dedicated server	N/A	N/A
Hummingbird [28]	n.c.	n.c.	Firefox plugin	dedicated server	$O(1)$	client
Persona [6]	$O(n)$	out-of-band	Firefox plugin	peers	$O(1)$	client
EASiER [53]	$O(1)$	n.c.	Facebook app.	peers	$O(n)$	proxy
DECENT [54]	$O(1)$	out-of-band	standalone	peers	$O(n)$	peers
Cachet [77]	$O(1)$	out-of-band	standalone	peers	$O(n)$	peers
Tran et al. [97]	$O(1)$	third-party	N/A	N/A	$O(n)$	proxy
Frientegrity [33]	$O(\log n)$	out-of-band	standalone	dedicated server	$O(1)$	client
PeerSoN [16]	PKI	out-of-band	standalone	peers	$O(n)$	client
porkut [76]	N/A	n.c.	N/A	peers	$O(1)$	client
Confidant [62]	$O(n)$	out-of-band	Firefox plugin	various	$O(1)$	client
Graffi et al. [43]	n.c.	implicit	standalone	peers	$O(n)$	client
PGP [17]	N/A	WoT	standalone	various	$O(n)$	client
OTR [2, 13]	N/A	SMP	standalone	N/A	N/A	N/A

Chapter 2

Proposed System architecture

In Chapter 1 we listed various approaches used for the realization of privacy-aware OSNs. For the realization of SNAKE we decided to create a new standalone FOSS OSN that employs the widely used client-server model. We did not choose to use neither the overlay solution nor the P2P one due to the problems highlighted during the study of the state of the art.

As shown in Figure 2.1 on page 20, there are three main components in our system:

Client The client is an HTML5 application designed to run in a web browser but easily portable to other environments such as a browser plugin and a desktop or a mobile application (thanks to technologies such as PhoneGap [15]).

Storage server A server where all the user data is stored running a very simple data providing service. The client interacts with the storage server through a HTTPS interface.

Application distribution server This component hosts the code of the client application, and serves it over HTTP to the end-user web browser over a Transport Layer Security (TLS) connection. His role is extremely simple: it just delivers static files, in particular it never interacts with the storage server, only the client does. In fact, it could even be a simple file server.

We proceed briefly illustrating some use cases and then describing each of the three components in detail.

Friends management A user can establish a new friendship which allows him to securely communicate with the new people. A new friend can be added to one or more groups he manages, in particular his friend's group. When a relationship with a user is terminated, the user is removed from the friend's group and his messages are ignored.

Exchange private messages A user can securely send and receive private messages to friends with whom a friendship is in place.

Manage groups A user from the client can manage groups. Groups can be groups of his own friends (which can be more than one) or generic groups of people such as «Students at Politecnico di Milano». The user can add another user to a group with different privilege levels. These privileges regulate if the participant can just read messages for the group, post new ones or also administer the group. Users can also be banned, which means they will not be able to read future messages or post new ones.

Exchange messages with a group The user, through the client, can read messages posted to the group and, if he has the proper privileges, post a new ones.

View recent posts aggregated The user, through the client, can also view in a single page the «stream» of all the recent messages that have been posted to groups he is participating to.

2.2 Analyzed scenarios

Before designing the detailed system architecture we defined the scenario in which we want to realize our application. This is a very important phase because the architectural choices of the system have to be made in relation to the considered scenario. With this analysis we can define a precise threat model for our application, which will be described from now on, to show which guarantees we can provide to the user. Note that not all the threats will be addressed since some of them simply do not depend on us and are therefore out of scope.

The storage server is used to store all the user's data and information exchanged between users within the OSN. The confidentiality, integrity and authentication of this data is already enforced by the client, but we must also

ensure the privacy of the associated metainformation. Despite the fact that every record stored by the server is encrypted, sensitive information might be derived anyways.

We do not want data on the storage server to leak information about users, friendship relations, group memberships and all the other types of interactions between users. This kind of attack can be carried on by the server itself, by someone collecting data of other users, or even by a government agency that seizes the data.

The type of information that can be derived strongly depends on the server. We will consider two different scenarios showing what are the threats and the guarantees over data of the users that we provide: a MALICIOUS- and an HONEST-SERVER scenario.

2.2.1 MALICIOUS-SERVER

In this scenario the storage server is untrusted in two aspects: he can tamper with the data and store additional metadata to track the user.

The first aspect considers all the various types of passive and active attacks aiming to break the secrecy of the exchanged messages, in particular we also consider MITM attacks.

The second aspect considers that the storage server can also try to track the user as much as possible. But this is not limited to the storage server, we also take into account the case where someone is sniffing the network traffic generated by a client. Information that may be stored along with each record are: the IP address that has generated an operation, the user agent of the browser used to connect to the web server, the operating system, the mobile device used, the timestamp of each network request and any other fingerprinting information that may be used to recognize a user and all the data he published.

Looking at the decorated data, it is possible to deduce information like: the user x uses our application only by its Android smartphone, x lives in Italy and uses our application twice a day, x has sent messages to the user y and so on. However, we will be able to guarantee that it is not possible to obtain the content of the exchanged messages or other data pertaining the system. Every other type of connection among users can only be guessed but not proven with the data.

2.2.2 HONEST-SERVER

In this scenario the storage server can be considered trusted. As in the previous model it offers the service regularly and fulfills all the requests. In addition to this we assume that the server does not store any kind of the aforementioned additional data.

In this scenario our system will guarantee that by looking only at the data *at rest*, no significant information can be derived. It will be only possible to understand if a certain user exists and match it with his public profile. In practice no relevant information is disclosed, except from what the user explicitly marked as public.

2.3 Application distribution server

The role of the application distribution server consists in delivering the client to the end user browser over HTTPS. However this is only one of the possible ways through which it is possible to obtain the client. For instance, if the client is in the form of a mobile application, in most cases it will be distributed and updated through a marketplace for the device platform. Another option is to install it as a web browser extension, in which case the browser will take care of updating it frequently from his own repositories.

If the application distribution server is used, it is a trusted component which has to be chosen carefully by the end-user since it can alter the application on the fly and completely compromise the user's privacy. It should be hosted by an entity which frequently updates, reviews, tests and then deploys the code. This entity should also properly manage its TLS certificates. For these reasons, the ideal solution is to have dedicated entities to handle this.

The reference model we have in mind are GNU/Linux distributions that digitally sign software packages, such as Debian [82]. Both in GNU/Linux distributions and in our case, an attacker trying to compromise a user account has to be able to actively manipulate the connection with the server hosting the application (i.e. carry on a MITM attack) and get access to the distribution server private key. In our case, the attacker needs the server's TLS private key to break the secure connection, while in Debian's case the attacker needs the GnuPG private key used to sign packages. There are however some differences. First of all a web page is updated more frequently than the whole system, in-

creasing the exposure to attacks, although in a frequently updated system the difference is negligible. Another issue is the fact that in case a compromised client application is used for a time lapse, and subsequently the uncompromised client is restored, no trace of the break-in remains, while GNU/Linux distribution usually keep a log of updates. Another difference lies in how frequently the private key is used: in a web server the TLS private key has to be always available to the daemon process, while the GnuPG private key is needed only upon a new package release and can therefore receive better protection.

In this work we assume TLS is secure and no one without the server's private key can alter the content of HTTP responses. In particular, problems deriving from the TLS PKI or Certification Authority (CA)s are out of the scope of our work. If this is an issue, we refer to other works such as Perspectives [101] and Convergence [70].

2.4 Data storage server

The storage server is responsible for storing all long term data and keeping it available from everywhere. It is composed by a thin HTTP interface realized in Node.js [25] and a RDBMS as a backend for data storage, specifically MySQL [23]. Note that in the following we consider the storage server as a single entity, since our work is more focused on the client. However nothing prevents us from building the storage server as a distributed system with horizontal partitioning of the data. Even better, the clients could connect to a storage server, part of a network of independent federated storage servers, which serves the data it has directly available or, if it is not, forwards the request to the appropriate storage server. We leave exploration in this direction for future works.

The storage server is considered untrusted in the MALICIOUS-SERVER scenario and trusted in the HONEST-SERVER scenario. In practice this means that we are able to guarantee the confidentiality, integrity and authenticity of the content of the exchanged messages even if the server is untrusted, but to obtain guarantees about privacy of the social graph, we have to assume the server stores only the information the client sends it. If the storage server stores additional metadata such IP addresses, browser characteristics or even timestamps, these guarantees may be lost. Note that if we consider the first scenario, the connection between the client and the server may also be unencrypted as the information we send is considered to be stored on an untrusted

storage provider. By contrast using an encrypted communication transport layer such as TLS (i.e. HTTPS) offers great benefits for the second scenario, as it protects our metadata from an third entity monitoring the connection, which could store the flowing information with richer metadata than an honest storage server would. Further dissertation on this in Chapter 5.

The storage server basically executes CRUD operations requested by clients. The only access control rule enforced by the storage server consists in allowing only the original creator of a record to remove or update it. This is possible since when a user asks the storage server to insert a new record, it also sends an `editToken` which is stored along with the record, and it cannot be read or updated. When the user is in need of deleting or updating the record, he will provide, as a proof of authorization, the `editToken`. Since the token is secret and only known to the original author he does not need any further authentication. From the server's point of view the `editToken` is simply a random bit sequence, however the client may use the digital signature of the record, encrypted with a dedicated key. Note that this type of per-request authentication, allows us to keep the protocol completely stateless. Note also that since a user is in full control of his own data, he may alter it at his will without the need to notify the other users. Besides this enforcement, the storage server executes any operation request by unauthenticated users: integrity and authentication checks are made on the client-side.

In Figure A.1 on page 133 we can see the tables composing the relational database on the storage server. Its structure is simple: a table for the user descriptors (`User`), a table for profile descriptors (`Profile`), a table for group descriptors (`Group`), a table for both private messages and posts to groups (`Message`) and a table for the Web of Trust (see Section 3.4). The four main tables, `User`, `Message`, `Group` and `Profile` are composed by a primary key, used by the client to query the database, the `editToken` described above, a series of public fields and a single container for all the encrypted fields, `privateData`. The `Group` table also contains an `adminData` field, which is similar to `privateData` but intended only for use by administrators of the group. We detail the meaning of the other fields in Subsection 2.5.2, while the `WoT` table is described in Section 3.4. For further information on how the primary keys are handled and on join paths refer to Section 5.2.

2.5 Client

The client is the component in charge of most of the whole system's logic. It receives, decrypts, checks integrity and authenticity of all data received from the storage server and presents it to the user. Viceversa, it also encrypts, digitally signs and sends to the storage server all the data generated by the user.

There are many threats to which the client is exposed. For instance, considering a web browser based client, it is possible that security bugs allow an attacker to compromise our application. If the client is compromised we cannot guarantee the confidentiality, integrity and authenticity of the information sent and read by the user. Therefore we assume that the client is running on a trusted system and runs applications that provide a safe, uncompromised environment. All the possible problems arising from the use of a compromised system or problems caused by security bugs of other applications will not be addressed as they go beyond the scope of this work. Given that the integrity of the client is not compromised by the platform it runs on, we are able to guarantee the confidentiality, integrity and authenticity of user's information.

2.5.1 Client architecture

In the simplest configuration, the client consists in a Single Page Application [39] which the user downloads from the application distribution server through his web browser. It is composed by HTML5 markup, JavaScript code and CSS stylesheets.

Following the best practices in JavaScript development, all of our code is asynchronous. This implies that the user interface remains responsive while network and cryptography related tasks are running in background, and is notified of their completion through callbacks. To use in an elegant way the asynchronous paradigm we use DOM4's Promises [11] with the support of a polyfill [89].

The application is designed following the Model-View-Presenter (MVP) paradigm and it is entirely based on Backbone.js [3], a lightweight JavaScript framework in turn based on the Underscore.js library [4], which we substituted with a more consistent, better performing and compatible library: Lo-Dash [26].

Our implementation is divided in the following components, in large part inspired by the Backbone.js structure:

Snake.Model A model is a data container for an object of the system such as an user, a profile or a message. It agnostically holds a dictionary of key-value pairs called *attributes*, no structure enforcement is done at this stage. A model also contains default values for newly created models, references to connected models (e.g. profiles of a user), the logic to fetch them automatically, event management routines (e.g. «model ready» or «model modified») and the logic to obtain the associated signing, verification and encryption keys.

Snake.View A view is a component responsible for rendering a certain model in a certain context and for handling user interaction. When a view is instantiated it is connected to a model. It is possible to have more than a single view for the same model, for instance the profile model has two views. The first shows all the possible information for the profile page, while the second one, used to render user lists, just shows minimal information such as picture, name and surname.

To elegantly handle the rendering of model data in views we used the Transparency [79] and jQuery [87] libraries.

Snake.Router If the URL of the application changes, the router intercepts the event and handles it without reloading the page or making an HTTP request. The router works with the URL fragment identifier. This part of the URL is usually used to identify a particular point a web page and it is not sent to the server. In our case we use it to redirect the user to a new view. In our system the URL of user profile with ID 4 looks like this:

`https://distributionserver.com/snake.html#profile/4`

When the router parses this URL, it creates a new **Profile** model. The newly instantiated model launches a series of requests to the remote storage to fetch the required data to populate itself and the connected models. In the specific case, the **Profile** model is connect to the **User** model of its owner, the associated **Group** models which in turn are connected to

the **User** models of their members. Then it creates a new **Profile** view and associates it with the **Profile** model. Once the model is ready, the view takes care of showing it to the user.

Snake.SocketIO This is the network component. It takes care of serializing, encrypting, signing and sending to the storage server outgoing objects, and, symmetrically, receiving from the storage server, decrypting, verifying the signature and deserializing incoming objects. It also handles creation and computation of **editTokens** upon creation and modification of a record.

To communicate with the storage server we use the Socket.IO [86], a library mainly built over WebSockets [46], a standard protocol providing full-duplex communication over a single TCP connection designed for interactive communication between web browsers and web servers. Socket.IO offers the same features but adds several fallback backends in case WebSockets are not supported by the user's web browser.

We opted for Socket.IO because it allows simple interactive communication with extremely low overhead if compared with plain HTTP. Interactivity is required for an application where almost all the system intelligence is on the client, and it reveals particularly useful for the chat application.

Snake.Crypto A wrapper around the cryptographic primitives (the WebCrypto API, see Subsection 2.5.3) to ease their usage from the rest of the application.

Snake.Schema The Snake.Schema component is used by the network layer (Snake.SocketIO) to validate incoming and outgoing objects. Since the internal representation of the models is in JavaScript Object Notation (JSON) format, we use JSON Schema, a standard, currently in draft state [36, 37]. JSON Schemas are defined in JSON itself and describe rules to validate a JSON document. Our validation is quite strict, and discards the JSON objects failing to pass it. This way after validation we can treat the data as correct without further checks. We chose to use the tv4 implementation [66] of JSON Schema.

Table 2.1: Summary of the models

Model	Encryption key	Access	Signing key	Table
SessionUser	User master key	Owner	Owner	User
User	N/A	Everyone	Owner	User
Post	Group shared key	Group members	Author	Message
PrivateMessage	Friendship key	Sender and recipient	Sender	Message
Profile	Profile key	Authorized friends	Owner	Profile
Group	Group shared key	Members	Group private key	Group
GroupAdmin	Group admin key	Administrators	Group private key	Group

2.5.2 Class diagram for models

As shown in Figure A.2 on page 134 and in Figure 2.1, the system entities are represented in the client with the following models:

User The **User** model is how a user is seen by other users, i.e. a collection of his public fields, in particular his user name, a reference to his public, unencrypted profile, the public key he uses for digital signatures plus a set of other public keys serving to the purposes of friendship establishment.

SessionUser This model inherits from the **User** model (in fact they come from the same database table) but adds all the fields which are only of interest of the represented user. The only additional relevant public field is **salt**, which stores the salt used for the password-based derivation of the user's master key upon login. Besides this we have a series of private fields holding all the private information the user needs. Three of them hold all the private keys associated to the above mentioned public keys. Then we have the list of user's friends along with the keys used to communicate with them on which the user agreed upon during friendship establishment. Moreover, for each friend the hash of his public data (see the **User** model) is stored along with a list of profiles the user has access to with the relative keys. A similar list is present for the groups, storing an hash of the group public fields, the symmetric key used to read and write to the group and, if the user is an administrator of the group, the private key of the group and the symmetric key for the administrative data (see the **GroupAdmin** model).

Profile A **Profile** is a simple container for personal information about a user, protected by an encryption key which is made available to a set

of authorized friends. The profile also holds a reference to the groups of which the owner is part. A **User** can have multiple profiles exposing different information to different (group of) friends. By default there is always an empty public, unencrypted profile. If a friend of a user has access to multiple profiles, he can choose which one to show or rely on the priority assigned by the owner.

PrivateMessage A **PrivateMessage** consists in a message sent directly by a user to another one. It can be a service message, for instance to handle a new friendship, or an actual text message. Information on how the sender and recipient identifiers are handled can be found in Chapter 5.

Group A group is the entity which allows many-to-many communication in our system. It can represent both group of friends of a specific user, if the **friendsOf** attribute is present, or more generic groups of people such as «Students at Politecnico di Milano». A group has a keypair used to sign the model, whose private part is given only to administrators.

Participants are divided in three categories depending to their authorizations: subscribers, members and administrators. Subscribers can only read messages of the group, members can also post new messages and administrators have access to the private key of the group and can therefore change group attributes such as description and the participants lists. Note that all the three types of participants can technically write new post or modify the group attributes, since they are all encrypted with the shared group key, but the various policies are enforced client-side through digital signatures. This means that if a subscriber posts a message in a group, the client will discard it since it is signed by a user not authorized to write to the group. In the same way, if a member is able to alter a group attribute, for instance bypassing the **editToken** protection colluding with the server, the clients will discard that group descriptor since it is not signed with the group private key.

When a user is removed from the group, an **endDate** attribute is added to his entry in the participants list, in this way other users can determine until which date he was allowed to post and therefore show or discard his messages appropriately. Moreover, when the shared group key is changed, it is made available to all participants except him, see Chapter 4 for further details. The **Group** model holds a list of all the previously

used keys along with their validity range.

GroupAdmin The **GroupAdmin** model is a support model encrypted with a key known only to administrators: it holds a support data structure for efficient key distribution upon rekeying, see Chapter 4.

Post A post is similar to a **PrivateMessage**, but it is addressed to a group instead than to a single user. It can be a service message, for instance to update the group key, or a real text post for the group. As for private messages, we defer the discussion about the handling of destination group and author identifiers to Chapter 5.

2.5.3 Cryptographic primitives

A fundamental enabling technology for our approach is the availability of in-browser cryptographic primitives: the WebCrypto API. The Web Cryptography API is a W3C standard [24], currently in draft state, describing a JavaScript API to perform basic cryptographic operations. It offers a series of primitives to compute digest, sign, verify, encrypt and decrypt data and to derive, generate, import and export keys. Several well known and standard algorithms are supported, such as RSA, ECDSA, the D-H key exchange and its version relying on elliptic curves (ECDH), AES with different operation modes, HMAC, SHA-1, the SHA-2 suite and some key derivation functions such as PBKDF2.

The WebCrypto API is of particular relevance in our project since it allows us to exploit the browser's underlying natively implemented cryptographic library and their PRNG, in most cases OpenSSL [84], NSS [75] or the Microsoft CryptoAPI [21]. Native implementations are critical not only for their enhanced performance figures but also because they are sound, well tested and have been thoughtfully reviewed over the years, in particular the open-source solutions. The WebCrypto API also offers an interesting feature: unextractable keys. An unextractable key is a key that is generated, imported or derived and used for various other operations, but cannot be exported. This is of particular use for instance to protect the user's master key. In fact, when a user logs in, his master key is derived from his password, and can be used with in several cryptographic operations but, if it is marked unextractable, it is not possible for the JavaScript code to get direct access to it. This means that if

Table 2.2: Current status of WebCrypto API implementations

Browser	Version	Missing primitives
PolyCrypt [8]	N/A	None
NfWebCrypto [51]	N/A	None
FoxyCrypt [7]	N/A	AES-GCM, ECDSA, ECDH
Internet Explorer [22]	11 ¹	ECDSA, ECDH, PBKDF2
Chromium [81]	33	RSA, ECDSA, ECDH, PBKDF2

the client is not compromised at login time, the key will always and only be used behind the scenes thanks to WebCrypto API. Note however that this is only true for the master key, since all other keys are stored remotely and are therefore handled directly in JavaScript.

Since WebCrypto API is an emerging standard in our prototype implementation we initially used a polyfill implementation. We used PolyCrypt [8], implemented and tested missing algorithms and kept it up-to-date with the evolving specifications. While this worked as a temporary solution for testing purposes, waiting for web browser developers to implement them, it was not suitable for realistic performance evaluations. For this reasons we took another implementation of the WebCrypto API, NfWebCrypto from Netflix [51]. NfWebCrypto is a Google Chrome/Chromium plugin, written in C++, exposing the WebCrypto API to our application using OpenSSL as a backend. In this case too we had to implement and test missing algorithms and keep the plugin synchronized with the specifications. In Chapter 7 we compare the two implementations. In Table 2.2 we show the current browser support for WebCrypto API.

As for the choice of cryptographic algorithms to use we opted for: ECDSA using curve `secp256r1` (also known as NIST P-256) for digital signatures, AES-256 in GCM or CBC mode for symmetric encryption, SHA-256 for hashing, HMAC with SHA-256 for MAC computation and PBKDF2 with SHA-256 and 1024 iterations for password-based key derivation. From now, if not differently specified, when we say «hash», «encrypt» and so on we refer to these algorithms. For the sake of simplicity, the system assumes these algorithms have been used but in future we might consider to specify in each record what algorithms have been used.

We chose to use ECC for its performance, reduced key size and security

properties compared to RSA. As for the elliptic curve to use, the WebCrypto API supports three curves over prime fields defined by the NIST in [58]: `secp256r1`, `secp384r1` and `secp521r1`. We opted for the first one since it has very small keys and a large enough security margin, roughly comparable to a 3072-bit RSA key. Since the only asymmetric algorithm supporting encryption and decryption in the WebCrypto API is RSA, we designed our system using only symmetric algorithms for encryption and decryption.

All the cryptographic primitives we used are supported by the WebCrypto API, while the employed protocols, FHEMQV-C and SMP (used in friendship establishment, see Chapter 3), are implemented in pure JavaScript. While they obviously achieve worse performance if compared with native implementations, we do not deem this as a real issue since the amount of computation is very small and spans over several network interactions which results in a negligible delay for the end user.

2.5.4 Object serialization process

When a model is saved, before being sent to the server it passes through a series of transformations. To efficiently store the data we employ the following libraries:

RJSON RJSON [30] is a library that compacts JSON objects reducing their redundancy. It takes a JSON object, and performs a lossless entropic encoding through a dictionary approach on the JSON structures. This allows to save a considerable amount of space through avoiding the repetition of key names in an array of homogeneous objects.

MessagePack MessagePack [38, 98] is an efficient way to binary serialize JSON objects. We use it to serialize data before performing cryptographic operations since they can only handle binary data. We chose MessagePack instead of its main competitor, BSON [50], because it is designed to be efficient in terms of size, which means less data to encrypt, decrypt, sign, verify and transfer over the network. We had to modify the JavaScript implementation to make it deterministic, as it employed the enumeration of the keys of a JavaScript object, which is not guaranteed to be preserved across instantiations, as an internal index. To overcome this issue we simply sorted the keys after enumeration.

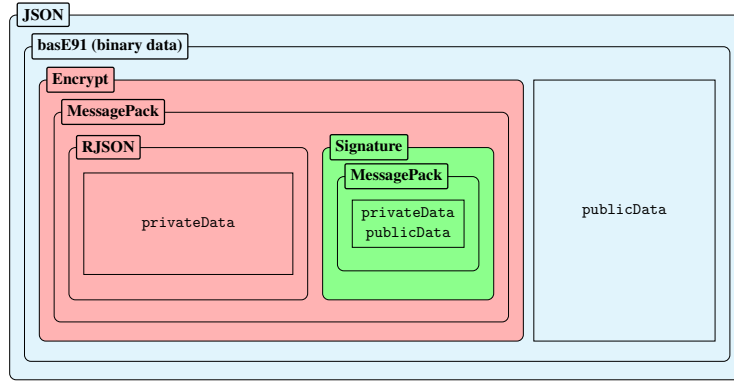


Figure 2.2: Scheme of the packing of a message before sending it to the storage server

basE91 Socket.IO works only with JSON objects, which does not support binary data. Therefore before passing the data to Socket.IO we serialize it in basE91 [45, 102], an encoding using only printable ASCII characters. We decided to use it instead of Base64 since it introduces a lower overhead, in fact basE91 increase the size of data of 23% on average, while for Base64 the overhead is 33%. We initially considered Base128, which uses all the 7-bits of the ASCII character sets, but had to switch away since JSON does not support single-byte encoding of non-printable characters.

As illustrated in Figure 2.2, the serialization process is composed of the following steps:

1. Take the input model attributes and split them in private fields and public fields accordingly to the schema of the model
2. Take the private fields and compact them with RJSON
3. Take all the model attributes, serialize them in the MessagePack format
4. Sign with ECDSA the serialized model attributes with the appropriate private key
5. Take the RJSON compacted private data and wrap it in a new object with the signature
6. Serialize in MessagePack the resulting object
7. Encrypt with AES in GCM mode the serialized object with the appropriate key
8. Wrap the resulting binary data in an object along with the IV used for

encryption

9. Take the result and serialize it in JSON transforming the binary fields in basE91
10. Take the resulting object and add it as **privateData** to an object containing the public fields
11. Send the object to the storage server

Viceversa, when an object is fetched from the storage server, the inverse operations take place in reverse order to retrieve the data.

Note that AES in GCM mode, along with confidentiality, offers data integrity. It also allows to specify additional unencrypted data to include when computing and verifying the integrity token. We pass all the public fields as additional data. This integrity check is a second-line defense, since we already digitally sign the data, but in future we might add models which are not signed through the external digital signature (see Chapter 8).

Chapter 3

Friendship establishment

In this chapter we describe the protocol used in SNAKE to establish a friendship and agree on a symmetric key that will be used to protect one-to-one communications among two friends. Since we use PKC, we also illustrate our two in-band approaches for public key authentication: the Socialist Millionaire Problem (SMP) approach and the Web of Trust (WoT) approach.

In the following we first introduce some notation and the MQV protocol and its derivatives, then we illustrate the whole protocol implemented in SNAKE for friendship establishment and we conclude describing how Web of Trust works in our system.

3.1 Notation

Let \mathbb{G} be the support set of a finite field of order q . \mathbb{G}^* is the support of the multiplicative group of the field, and let g be one of its generators. We denote $|x|$ as the minimum amount of bits required to represent $x \in \mathbb{G}$, i.e. $|x| = \lceil \log_2 x \rceil$. We denote public keys with upper case letters and the private keys with the corresponding lower case letter (e.g. A and a). Public keys are elements in \mathbb{G}^* , while private keys are elements in \mathbb{Z}_q . We denote as \hat{A} the identity of the owner of the keypair (A, a) and the user himself. We assume \hat{A} contains A or sufficient information to uniquely identify it. We denote with $x \in_R [a, b]$ the fact that x is picked uniformly in the range $[a, b]$.

Table 3.1: Comparison of the variants of the MQV algorithm. l is $\frac{|q|}{2}$. \bar{H} is an hash function producing $\frac{|q|}{2}$ bits of output, and H an hash function producing the desired number of bits for the session key.

Variable	MQV	HMQR	FHMQR
d	$2^l + (X \bmod 2^l)$	$\bar{H}(X, \hat{B})$	$\bar{H}(X, Y, \hat{A}, \hat{B})$
e	$2^l + (Y \bmod 2^l)$	$\bar{H}(Y, \hat{A})$	$\bar{H}(Y, X, \hat{A}, \hat{B})$
K	$\sigma_{\hat{A}} = \sigma_{\hat{B}}$	$H(\sigma_{\hat{A}}) = H(\sigma_{\hat{B}})$	$H(\sigma_{\hat{A}}, \hat{A}, \hat{B}, X, Y)$ $\quad \quad \quad =$ $H(\sigma_{\hat{B}}, \hat{A}, \hat{B}, X, Y)$

3.2 MQV and its variants

FHMQR [91] is an evolution of the HMQR protocol [59], which in turn is derived from the MQV protocol [60, 72], an highly efficient authenticated key exchange protocol by Law, Menezes, Qu, Solinas and Vanstone based on the D-H key agreement. The three protocols basically share the same structure. Given a cyclic group \mathbb{G} of prime order q generated by g , \hat{A} and \hat{B} want to establish an authenticated shared key. \hat{A} has a private key $a \in_R \mathbb{Z}_q^*$, a public key $A = g^a$, and generates another ephemeral pair of public-private keys, $x \in_R \mathbb{Z}_q^*$ and $X = g^x$. \hat{B} also has a long-term and ephemeral public-private keypairs $(b, B = g^b)$ and $(y, Y = g^y)$ respectively. The ephemeral public keys are exchanged and each party of the communication fetches the long-term public key of the other end from a key repository. At this point both \hat{A} and \hat{B} are able to derive a shared authenticated key as follows:

$$\begin{aligned}
 r_A &= x + d \cdot a & r_B &= y + e \cdot b \\
 \sigma_{\hat{A}} &= (Y \cdot B^e)^{r_A} = & \sigma_{\hat{B}} &= (X \cdot A^d)^{r_B} = \\
 &= (g^y \cdot g^{b \cdot e})^{x + d \cdot a} = & &= (g^x \cdot g^{a \cdot d})^{y + e \cdot b} = \\
 &= g^{(y + b \cdot e) \cdot (x + d \cdot a)} & &= g^{(x + a \cdot d) \cdot (y + e \cdot b)}
 \end{aligned}$$

$$\sigma_{\hat{A}} = \sigma_{\hat{B}}$$

where the values of d and e vary depending on the chosen MQV variant

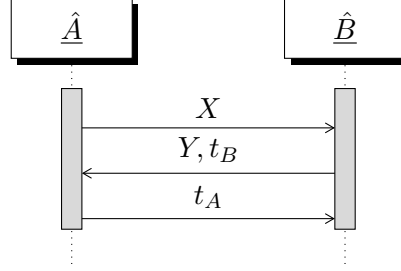


Figure 3.1: Exchanged messages in the FHMqv-C protocol

(see Table 3.1). Basically the plain MQV version just involves the ephemeral public keys, while HMQV mixes them with the identity of the users (\hat{A} and \hat{B}) through an hash function \bar{H} . FHMqv does the same but uses both identities and ephemeral public keys for both d and e .

The other significant difference among the three protocols resides in how the final key is generated: MQV does not enforce any specific transformation on the result of the protocol, while HMQV uses an hash function H on it and FHMqv does the same including in the hash the identities and the ephemeral public keys. The hashing function H is chosen to produce the desired key length.

All of the protocols have a version with key confirmation, which means that both parties have the guarantee that the other end computed the same key and was actually involved in the protocol. As depicted in Figure 3.1 we present a summary of the exchanged messages in the FHMqv version: FHMqv-C. To obtain key-confirmation two keys are generated using distinct key-derivation functions¹:

$$K_1 = \text{KDF}_1(\sigma_{\hat{A}}, \hat{A}, \hat{B}, X, Y)$$

$$K_2 = \text{KDF}_2(\sigma_{\hat{A}}, \hat{A}, \hat{B}, X, Y)$$

Both \hat{A} and \hat{B} compute a MAC, using K_1 as key, of their own ephemeral keys and identity, respectively:

$$t_A = \text{MAC}_{K_1}(\hat{A}, X)$$

$$t_B = \text{MAC}_{K_1}(\hat{B}, Y)$$

¹In our implementation it is an HMAC with SHA-256 using 0x01 and 0x02 as key

These values are then exchanged and checked by both parties for correctness. If the two keys match, K_2 is used as the session key.

3.2.1 Security features

HMQV has been proven to be resistant against a set of attacks in the Canetti and Krawczyk model [19]. In the following we present its main security features.

Simple impersonation attacks As shown in [59], MQV is insecure w.r.t. a simple impersonation attack because an attacker can impersonate \hat{A} to \hat{B} without the knowledge of a (\hat{A} 's private key). The attack is possible because d and e do not depend on the ephemeral keys, so an attacker can choose a specially-crafted value of X and impersonate \hat{A} .

HMQV solves the problem including the ephemeral keys in the computation of d and e .

Unknown Key Share (UKS) attacks We have a UKS scenario when \hat{A} and \hat{B} are able to compute the same key, which remains unknown to the attacker \hat{M} , but \hat{B} believes that on the other end of the communication there is \hat{M} and not \hat{A} . MQV was considered secure from UKS attacks if each party had to prove possession of the private key corresponding to the public key they were using, but, as shown in [56], it is not. In [59] has been proven that also the MQV variant with key confirmation is vulnerable to UKS attacks, in case of leakage of ephemeral session-state information.

To prevent this attack, (F)HMQV derives the session key using the identities of both the endpoints (\hat{A} and \hat{B}).

Key Compromise Impersonation (KCI) attacks A protocol is vulnerable to a KCI attack if recovering \hat{A} 's long-term private key a allows the attacker, not only to impersonate \hat{A} to another arbitrary user \hat{B} , but also to impersonate \hat{B} to \hat{A} . This is called *reverse impersonation*, and if a protocol is secure from this kind of attacks the only benefit gained in getting to know a user's long-term private key is being able to actively impersonate him, all the previous sessions established with uncompromised users remain secure.

As shown in [59], in MQV a KCI attack can lead to the discovery of the

session key of an uncompromised session in a session-key query scenario, i.e. if the attacker is able to retrieve the original session key of another handshake he is actively controlling. HMQV is not vulnerable to KCI in this scenario since enforces to hash σ before using it as a key, preventing any kind of computation on it in case of leakage.

Exposure attacks It has been proven that in HMQV, the shared session key is not compromised even if any pair of secret keys (a, b, x, y) is leaked, save for the (a, x) and (b, y) pairs, in which case the attacker knows everything about one of the users.

Key confirmation A protocol offers key confirmation if, after protocol termination, both ends know that the other end has computed the same key and actively participated in the protocol. This feature is offered by the “-C” variants of MQV (e.g. FHMQV-C). The main benefit of key confirmation is having a sanity check and moderating the impact of a Denial of Service (DoS) attack, since no further information is sent until each party is sure that the other end is actually able to decrypt it. This comes at the cost of an additional message to exchange.

Perfect Forward Secrecy (PFS) A protocol achieves PFS if, in case of leakage of all the long-term key material (i.e. long-term private keys), it is not possible to break the secrecy of a previous session. This property cannot be fully achieved with a 2-message protocol, since an attacker can always actively alter the communication injecting an ephemeral public key X^* (of which he knows the corresponding private key x^*) and afterwards, once \hat{A} ’s long-term private key a is recovered, recover the session key. HMQV provides a weaker form of PFS, i.e. only in case the attacker is passive during session key establishment. The variants of MQV with key confirmation (e.g. FHMQV-C) ensure that the other peer chose the D-H values, achieving full PFS.

Validation of public keys If public keys (ephemeral and long-term) are not validated an attacker could tamper with them. For instance, it is possible to force both parties to use a small group which would lead to a low-entropy key. In case the group \mathbb{G} is subgroup of a larger group \mathbb{G}' (as is typical in $\text{mod } p$), to enforce membership to the intended group \mathbb{G} , MQV authors suggested (without proving its safety) the exponentiation

to $h = \frac{|\mathbb{G}'|}{|\mathbb{G}|}$ in the final key derivation step, enforcing the result to be in the subgroup \mathbb{G} :

$$K = \sigma_A^h = \sigma_B^h$$

In HMQV the verification was not initially required, but was actually necessary as shown in [73, 74]. For this reason, FHMQV mandates public key validation. Note however that elliptic curves standardized by NIST have prime order and therefore groups defined over them do not have subgroups.

Universal composability (F)HMQV-C can be shown secure in the Universal-Composability (UC) model of Key Exchange [20], which means that it can be run concurrently with other protocols without losing any of its security features.

The FHMQV protocol offers the same security features of HMQV but is also proven to resist to ephemeral secret exponent leakage (r_A or r_B). Moreover FHMQV's proof of resistance to ephemeral private key leakage (x or y) relies on weaker assumptions, namely the Knowledge of Exponent Assumption (KEA1) [10] is not needed.

3.3 Our protocol

We divide the handshake to establish a friendship in 3 concurrent parts: a run of the FHMQV-C protocol to obtain an authenticated, confirmed shared key and two (optional) runs of the SMP described in [14] (proposed also for OTR in [2]) to mutually authenticate public keys. The authentication of public keys is optional since a user could trust the public key of the other end relying on an underlying Web of Trust (see Section 3.4).

Since our design is based on central server storing asynchronous messages between users, one of our objectives is to design a protocol which does not leave any explicit trace of who was involved in the handshake on the long-term storage. For this reason the identity of the initiator of the handshake is not explicit: \hat{A} 's identifier is sent in the payload of the message, encrypted with a key obtained by a simple D-H key derivation, K_0 . For an explanation of how message destinations are handled after the first message, see Chapter 5.

Each step of our protocol does some computation and produces some values to send to the other party, when all the values are ready the message is sent.

For our implementation we chose to use AES in CBC mode for symmetric encryption and decryption and, despite the multiplicative notation, the `secp256r1` (also known as NIST P-256) elliptic curve for asymmetric cryptography.

3.3.1 Variations on FHMV-C

Algorithm 3.1 on page 44 shows the contributions to the exchanged messages of the FHMV-C run. Steps marked in blue are the main variations introduced with respect to the plain FHMV-C protocol and consists in how \hat{B} gets to know \hat{A} 's identity and how \hat{A} gets to know \hat{B} 's ephemeral public key.

Step 1a shows the first relevant difference from FHMV-C. We decided to let each user \hat{B} publish in the key repository an ephemeral public key Y along with the long-term one (B). This way, another user \hat{A} willing to initiate an handshake can send right in the first message all the information needed to derive the session key and produce the key-confirmation value. This approach does not only make the protocol shorter, but also reduces the impact of DoS attacks, since \hat{B} can immediately know who is \hat{A} , and decide whether to accept or ignore the request without additional network or storage load on the system. However Y has to be actually ephemeral, therefore, as shown in step 2h, \hat{B} refreshes it each time the user notices a new handshake request. This means that an ephemeral public key can be used in several handshakes, but since the owner refreshes it as frequently as possible, and answers to all the handshakes using the same ephemeral key in the same moment, an attacker does not gain any advantage.

The second difference resides in steps 1f and 2c: identity of user \hat{A} is not sent in clear, but it is encrypted with a key K_0 derived through a D-H run using the temporary keypairs (i.e. (x, X) and (y, Y)). This way a passive attacker looking at the database after the ephemeral keys are expired is not able to understand who is involved in the communication, since he has access only to the encrypted value $I_{\hat{A}}$. To understand who is the sender, the recipient \hat{B} has to derive K_0 and decrypt $I_{\hat{A}}$. Note that an active attacker could perform a MITM attack which would disclose \hat{B} 's identity.

Algorithm 3.1 Variation of the FHMqv-C protocol

1. The initiator \hat{A} does the following:
 - (a) Fetch \hat{B} 's public keys Y and B from the key repository
 - (b) Validate Y and B
 - (c) Choose an ephemeral private key $x \in_R [1, q - 1]$
 - (d) Compute the ephemeral public key $X = g^x$
 - (e) Derive a key $K_0 = Y^x$
 - (f) Encrypt \hat{A} with K_0 producing $I = \text{Enc}_{K_0}(\hat{A})$
 - (g) Derive the keys K_1, K_2 using FHMqv-C with $(x, X, Y, a, A, B, \hat{A}, \hat{B})$
 - (h) Produce a confirmation token $t_A = \text{MAC}_{K_1}(\hat{A}, X)$
 - (i) Add (X, I_A, t_A) to message 1
 2. Upon receiving message 1, \hat{B} does the following:
 - (a) Validate X
 - (b) Derive $K_0 = X^y$
 - (c) Obtain $\hat{A} = \text{Dec}_{K_0}(I_A)$
 - (d) Fetch \hat{A} 's public key A from the key repository
 - (e) Validate A
 - (f) Derive the keys K_1, K_2 using FHMqv-C with $(X, y, Y, A, b, B, \hat{A}, \hat{B})$
 - (g) Check $t_A = \text{MAC}_{K_1}(\hat{A}, X)$
 - (h) Refresh Y
 - (i) Produce a confirmation token $t_B = \text{MAC}_{K_1}(\hat{B}, Y)$
 - (j) Add t_B to message 2
 3. Upon receiving message 2, \hat{A} does the following:
 - (a) Check $t_B = \text{MAC}_{K_1}(\hat{B}, Y)$
-

3.3.2 Public key authentication with SMP

The Socialist Millionaire Problem (SMP) basically ensures that both the ends of the communication know a common secret, revealing nothing about it save whether the secret is actually the same or not. This is possible since to check whether an attempt to answer was correct or not, an interaction from the other end is required. In our case we call the secret value o when \hat{A} is computing it and p when \hat{B} is doing it. o and p are computed as the hashes of the key confirmation values t_A and t_B and the identities of \hat{A} and \hat{B} , salted with the hash of the answer to a question C . The question C is what allows to authenticate the hashed information: its purpose is to recall a pre-shared secret \hat{A} and \hat{B} previously agreed upon. However it is not necessary to have an explicit agreement, the question might be about a fact that only the two parties know about. If the attacker does not know the answer to the question he will not be able to produce the correct value and the authentication will fail.

In the following and in Algorithm 3.2 on page 57 we describe step-by-step our implementation of the SMP.

Step 1 In the first step \hat{A} uses g_1 , a fixed generator of \mathbb{G} , to generate two keypairs, (U_2, u_2) and (U_3, u_3) , and chooses a question C for \hat{B} . The public keys and the question are sent in message 1 to \hat{B} , encrypted using the FHEMQV-C derived key K_2 .

Step 2 Upon receiving and decrypting message 1, \hat{B} validates the public keys, generates with g_1 two keypairs, (V_2, v_2) and (V_3, v_3) . \hat{B} uses his private keys and the public keys of \hat{A} to derive with D-H two keys: g_2 and g_3 . \hat{B} answers the question chosen by \hat{A} and computes its hash $K_{\hat{B}}$. At this point \hat{B} derives the shared secret o computing the MAC, keyed with $K_{\hat{B}}$, of his own public descriptor, the public descriptor of \hat{A} and the FHEMQV-C confirmation tokens t_A and t_B . \hat{B} picks a random v and computes the values $P_v = g_3^v$ and $Q_v = g_1^v g_2^o$. The two public keys, P_v and Q_v are encrypted with K_2 and sent in message 2.

Step 3 Upon receiving and decrypting message 2, \hat{A} validates the received public keys, derives g_2 and g_3 , gives his answer to C (which might be different from the one of \hat{B}) and computes, in the same way \hat{B} did, the shared secret p . He picks v at random and computes the values $P_u = g_3^u$,

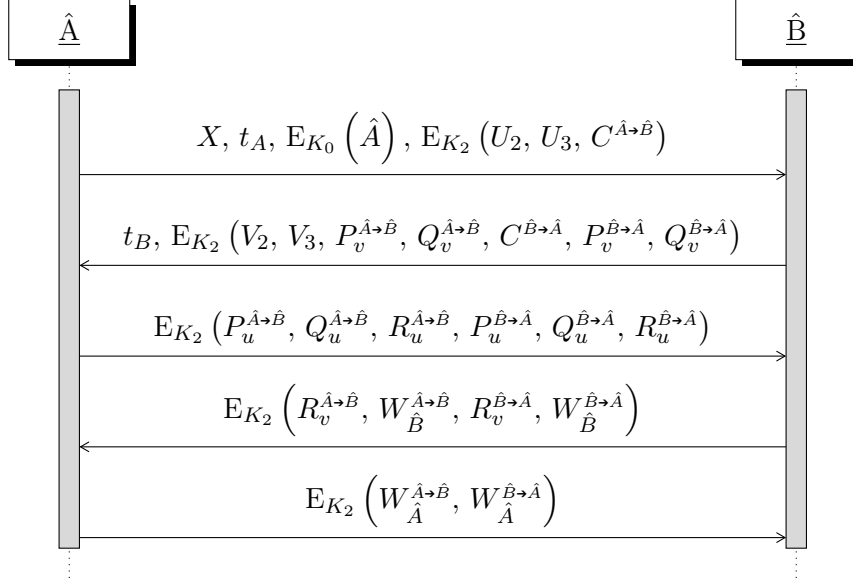


Figure 3.2: Messages exchanged between the parties in the final protocol. $E_y(x)$ means that value x has been encrypted with key y . $x^{\hat{A} \rightarrow \hat{B}}$ means that value x belongs to the SMP run in $\hat{A} \rightarrow \hat{B}$ direction, viceversa for $\hat{B} \rightarrow \hat{A}$.

$Q_u = g_1^u g_2^p$ and $R_u = \left(\frac{Q_u}{Q_v}\right)^{u_3}$. P_u , Q_u and R_u are encrypted with K_2 and attached to message 3.

Step 4 Upon receiving and decrypting message 1, \hat{B} computes $R_v = \left(\frac{Q_u}{Q_v}\right)^{v_3}$ and checks if the shared secret matches, that is if $o = p$. He does so verifying that $R_u^{v_3}$ and $\frac{P_u}{P_v}$ have the same value. Then \hat{B} encrypts with K_2 and attaches to message 4 the value R_v and a value $W_{\hat{B}}$ with the result of the comparison.

Step 5 Upon receiving and decrypting message 4, \hat{A} checks $o = p$ verifying if $R_v^{u_3}$ equals $\frac{P_u}{P_v}$. Then \hat{A} encrypts with K_2 a value $W_{\hat{A}}$ with the result of the comparison.

For a detailed proof of soundness and correctness of the protocol is available in [14].

3.3.3 Composing the protocol

The final protocol is composed by a run of FHMV-C and two runs of the SMP. We use FHMV-C to obtain the confirmation tokens t_A and t_B

Table 3.2: Resulting log on the database. The first two columns indicate the index and the direction of the message. The *Message* column represents the payload of the message, divided per protocol run. *Key* indicates the key used to encrypt the corresponding part of the message.

#	Direction	Message			Key
		FHMVQ-C	SMP		
			Common	$\hat{A} \rightarrow \hat{B}$	
1	$\hat{A} \rightarrow \hat{B}$	X, t_A			plain
		\hat{A}			K_0
		U_2, U_3	$C^{\hat{A} \rightarrow \hat{B}}$		K_2
2	$\hat{B} \rightarrow \hat{A}$	t_B			plain
		V_2, V_3	$P_v^{\hat{A} \rightarrow \hat{B}}, Q_v^{\hat{A} \rightarrow \hat{B}}$	$C^{\hat{B} \rightarrow \hat{A}}, P_v^{\hat{B} \rightarrow \hat{A}}, Q_v^{\hat{B} \rightarrow \hat{A}}$	K_2
3	$\hat{A} \rightarrow \hat{B}$		$P_u^{\hat{A} \rightarrow \hat{B}}, Q_u^{\hat{A} \rightarrow \hat{B}}, R_u^{\hat{A} \rightarrow \hat{B}}$	$P_u^{\hat{B} \rightarrow \hat{A}}, Q_u^{\hat{B} \rightarrow \hat{A}}, R_u^{\hat{B} \rightarrow \hat{A}}$	K_2
4	$\hat{B} \rightarrow \hat{A}$		$R_v^{\hat{A} \rightarrow \hat{B}}, W_{\hat{B}}^{\hat{A} \rightarrow \hat{B}}$	$R_v^{\hat{B} \rightarrow \hat{A}}, W_{\hat{B}}^{\hat{B} \rightarrow \hat{A}}$	K_2
5	$\hat{A} \rightarrow \hat{B}$		$W_{\hat{A}}^{\hat{A} \rightarrow \hat{B}}$	$W_{\hat{A}}^{\hat{B} \rightarrow \hat{A}}$	K_2

and to agree on a shared key K_2 . SMP is used to authenticate public keys, if necessary. We need two parallel runs of SMP to allow both ends to choose a proper question. If we do not do this, an attacker, Mallory, might initiate a friendship handshake with Bob pretending to be Alice and, to persuade Bob he is Alice, he could ask an unsafe question. In particular, Mallory will choose question whose answer is not known only to Bob and Alice, but also to himself. If Bob does not deem the question appropriate, he may be willing to ask another question to Alice (Mallory) through a SMP run in the opposite direction.

Table 3.2 shows what appears on the database after an handshake, while Figure 3.2 on page 46 summarizes the exchanged messages. Note that, the two pairs of public key used in SMP, U_2, U_3, V_2 and V_3 can be shared among the two runs without any loss from the security point of view, since the keypairs cannot be reused in future sessions.

3.3.4 Final key derivation

To prevent splicing attacks and be completely sure that messages have not been altered, the final session key is not simply obtained from K_2 but involves all the messages of the handshake. Both \hat{A} and \hat{B} before sending and upon receipt of a message update a value J associated with the ongoing handshake as follows:

$$J_i = \text{MAC}_{J_{i-1}}(m_i)$$

where i indicates the index of the current message and m_i the message itself. The last step consists in computing the final key as $K = \text{MAC}_{K_2}(J_5)$.

3.4 Web of Trust

The use of the SMP to authenticate public keys allows us to avoid out-of-band communication exploiting pre-existing implicit secrets among users. Although asking a question is not a big overhead for the end user, we want to keep interaction with the user during friendship establishment to the minimum. For this reason we introduced another way to authenticate public keys based on the concept of Web of Trust (WoT), as an alternative to SMP. Avoiding usage of SMP not only reduces the burden on the user but also makes friendship establishment shorter, since only FHMV-C has to be run. This leads to 3 message exchanges instead than 5.

The idea behind the Web of Trust is to exploit existing relationships to establish new ones quickly. Suppose Alice has recently subscribed to SNAKE but already has a couple of friends, including Charlie and Dorothy. Basically, we use existing secure communication channels with current friends (Charlie and Dorothy) to check whether any of them can confirm that the public key of Bob actually corresponds to him (because they already verified that with a certain degree of trust). If she wants to add Bob as a friend, who is already a friend of Charlie and Dorothy, she can avoid to use the SMP to authenticate Bob's public key and ask their two common friends if they have the same public key. The request is actually intended as a check on Charlie's and Dorothy's friend lists: these lists contain the user ID of the various friends along with the hash of their public descriptor, which includes the public key. If Alice can find enough common friends (more than a fixed threshold) she can mark Bob's

Table 3.3: Summary of the different approaches to set up the Web of Trust. *Query cost* indicates the cost of a query w.r.t. in the size of the group n in terms of network traffic generated. The last two columns indicate whether false negatives/positives are possible and the reason.

Method	Query cost	False negatives	False positives
Fetch everything	$O(n^2)$		
Recommendation	$O(n)$	Malicious user	
Reduced representation	$O(n^2)$		Bloom filter
Dedicated table	$O(n)$	Malicious member + storage server	

public key as authentic.

Note that in the following we will use interchangeably the terms «group» and «friends of» since the friends of a user are modeled as a group.

3.4.1 Querying the WoT

In Chapter 1 we briefly described the original and most popular existing Web of Trust: PGP’s Web of Trust. One of the problems with PGP is that the WoT is public and therefore leaks a certain amount of sensitive information such as user relationships, not just to the key sever, but to the world. For this reason we designed a concept similar to a Web of Trust, but private per user.

Being private is an important feature, but poses a number of issues when a user has to query it since the key server cannot index or perform searches over encrypted data. For this reason we need an efficient way to perform a query, i.e. discover rapidly if and who among Alice’s friends is friend with Bob. We came up with four approaches summarized in Table 3.3 and described in the following.

Fetch everything The naïve approach consists in letting Alice fetch all the lists of her friends’ friends looking for Bob ID. This method is 100% reliable (i.e. if we have a friend in common with Bob, Alice will find it) but unpractical. In fact if we consider an average of 500 friends per user (which may sound conservative but depends on the considered population, see [69]), we would have to collect information about $500^2 = 250000$ friends of friends IDs and decrypt them. If we consider even as little as 32 bit per user that would result in an amount of data in the order of megabytes.

Suggestion from Bob Another option is to let Bob suggest a list of common friends. Bob, to do this without checking all his friend of friend's list, could simply send Alice a list with all the IDs of his friends² (or more precisely a list of the groups he is in). Alice would then compute the intersection set between this list and her list of friends, obtaining the set of common friends. The suggestion would allow Alice to query only the strictly necessary friends but it also poses a security risk. In fact Bob could omit on purpose some friends that could unmask him as an impostor.

Reduced representation An alternative we considered is to let each user offer a reduced representation of his own friends list. This representation should be kept up to date by the user, say Charlie, and fetched by a friend, Alice, when she needs to authenticate Bob's public key. After testing several compression modes without success (mainly due to the fact that we assume user IDs are uniformly distributed over the ID space, see Chapter 5), we found that the most effective approach is using a Bloom filter [12]: using just 11 bit per user we can get false positive rate of 1%. Bloom filters have 100% reliability since they do not have false negatives. However false positives result in unnecessary network traffic which gets added to the 11 bits-per-user.

Dedicated table The last option we explored consists in the creation of a dedicate table on the server-side. This table should allow to test whether a user is a member of a certain group or not, leaking as little data as possible about the associated group. The table should be updated by a group administrator each time a user is added or removed. While this solution introduces an additional overhead for the administrator, it keeps the cost membership test for the members very low. In practice when Alice wants to know what groups she shares with Bob, she has to query the table once per group.

²Actually this would disclose Bob's friends to Alice (who has not been authenticated yet). To overcome this problem Bob could send an alternative ID of each friends, known only to the friends of that friend.

3.4.2 Web of Trust table

We opted for the last approach. As said the table should leak as little information as possible to the storage server, keeping this in mind we designed the WoT table with one record per member per group. As shown in Figure A.1 on page 133, the WoT table contains the following fields:

member This is the field used by the user to make queries. It contains the MAC of the member's ID using the group key as key.

group The MAC of the group ID using the group key as key. This value can be computed by each member of the group.

editToken A token known only to the administrators of the group, used to delete the row and to authenticate the administrator when adding a new member. As all the other edit tokens (see Section 2.4) it is write-only and therefore it is not returned to the user as query result. This token is shared for all the rows related to the same group.

When an administrator changes the group key, usually due to a removed user, he first drops the old list launching a purge request to the server with the **editToken**. Since the **editToken** is shared among all the rows of the group, the administrator is able to remove all the rows with a single message. In the same request to the server, the administrator sends the new list of members using the new key. If the administrator wants to add a new member, he does not need to create a completely new list (since the key did not change) but he has to prove the server he was the creator of the original list sending the **editToken**. To ensure only the administrator can add members, the server will refuse to save a row with a group identifier already present but with a different edit token. For this reason, when a rekeying takes place, the administrator must update the WoT table before notifying members of the new key.

When Alice wants to know if Bob is in a group managed by Charlie, she has to send the server a request with the computed **member** value and nothing else. The response will contain the match result and in the positive case it will also contain the associated **group** value. Alice can then check that the row was submitted with the right **group** and not a fake one to bypass the above mentioned server side check. With one of these requests for each group she belongs to, she can get to know all the groups shared with Bob, fetch

their descriptor with the full member list and check if the hash of Bob's public descriptor matches.

We would like to highlight that while describing the WoT table we never talked about signatures: we avoid them because they would be an overhead to gain a small benefit. In fact, due its structure, and with the help of the server, only the administrators can modify the group's WoT data. If a member and the server collude, they could introduce false negatives and false positives. However, this is a marginal problem since false positives would just generate unnecessary traffic and false positives are not critical since the WoT is just one of the two means to authenticate public keys: even if the server completely disables the WoT, Alice will always have the chance to authenticate Bob's public key through the SMP method.

As a last point, an attacker with access to the database at rest, looking at the WoT table, is not able to get any useful information since all the fields are hashed using a secret key or are just random data. The only useful information revealed is the existence of a group with a certain size, since rows can be grouped together by the `group` or the `editToken` field.

3.4.3 Trust level computation

To take advantage of the Web of Trust we need a way to evaluate the information it gives us. In particular we need an algorithm to compute a trust level for the new friend Bob and set a threshold above which the Bob's public key is deemed trustworthy.

3.4.3.1 Trust computation in GnuPG

In this work we adapt the approach used by default by GnuPG [83], a popular FOSS implementation of the OpenPGP specifications [17].

In GnuPG the Web of Trust is a graph where the nodes are the users (along with their public keys) and the edges are signatures over the public key of a user by another user. A key can have one of the following trust states depending on how trustworthy the owner is:

Ultimately trusted State reserved for the user's keys.

Fully trusted The owner has an excellent understanding of key signing, and his signature on a key would be as good as your own.

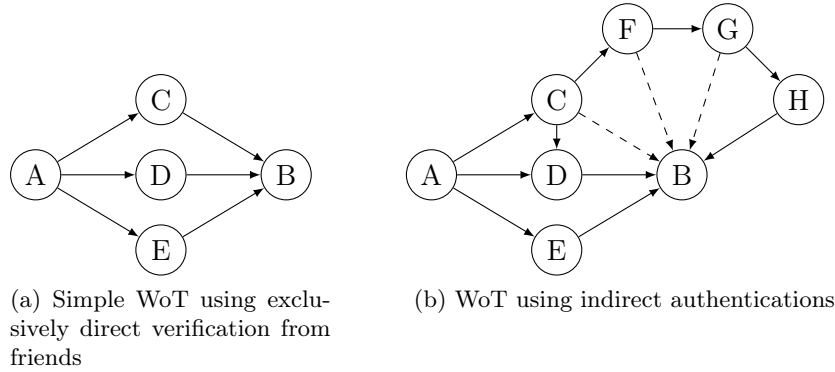


Figure 3.3: Web of Trust examples. A node is a user, an edge represents a SMP verification.

Marginally trusted The owner understands the implications of key signing and properly validates keys before signing them.

Unknown No trust evaluation has been assigned.

None The owner is known to improperly sign other keys.

These states or not part of the Web of Trust, they are private evaluations done by the user.

When using the WoT, by default, a key is deemed correct if it has been obtained through a chain of signatures shorter than five steps and if one of the following conditions are met:

- the key has been directly signed by the user;
- the key has been signed by at least one fully trusted user;
- the key has been signed by at least three marginally trusted users;

3.4.3.2 Trust computation in SNAKE

Our approach take heavy inspiration from GnuPG WoT. A node in our Web of Trust is a user, while an edge is a friendship relation verified through a SMP run. For the sake of simplicity, we do not differentiate trust levels at this stage: to consider a public key authentic three confirmations are needed. In Figure 3.3a we can see the simplest situation for a successful authentication through the WoT: Alice wants to verify Bob's public key, and they have three

Table 3.4: List of friends of Alice’s friends of the scenario in Figure 3.3b on page 53

User	Friend list
Erin	B
Dorothy	B
Charlie	D
	F
	B,[(D, 1), (H, 3), ...]
Frank	G
	B,[(H, 2), ...]
George	H
	B,[(H, 1), ...]
Henry	B

friends i common Charlie, Dorothy and Erin. All three of them have verified Bob’s public key through a SMP run, and Alice gets to know this checking their friends list.

In Figure 3.3b on page 53 we have a slightly more sophisticated situation. In this case Charlie is friend with Bob, but he did not verified his public key directly: he used the WoT through Dorothy, Frank and a third user not represented³. Frank is friend with Bob but used the WoT through George, which in turn verified the public key through Henry. Henry verified the key with SMP. Alice needs to collect information about at least three verification with SMP, paying attention to not count twice a single verification, for instance the $D \rightarrow B$ verification should be counted only once, even if there are two paths from Alice to Bob passing through Dorothy ($A \rightarrow D \rightarrow B$ and $A \rightarrow C \rightarrow D \rightarrow B$). For this reason, as illustrated in Table 3.4, in Charlie’s list of friends we do not simply expose the fact that he knows about a certain number of SMP verifications, but we also store how long and who is the last step of the chains leading to Bob, in this case Dorothy at distance 1 and Henry at distance 3. Once Alice has collected all the necessary information, she can verify how many distinct verification less than five steps away she saw, without counting multiple times the same verification. Notice that in case one of Charlie, Dorothy and

³From now on we ignore the fact that three verifications are needed to confirm the public key, except for the user we are focused on: Alice.

Erin was malicious, there would be no advantage in lying about the amount and the distance of the verifications of Bob's public key: the most effective attack would be simply marking Bob's as directly verified through SMP.

Since we do not want the WoT to leak information unnecessarily, in the friend list we do not store the real identifier of the verifier, but an associated unique ID known only to those who are friends with him. This way we are still able to avoid to count twice the same verification without leaking information.

While the described approach works in the trivial case, if we introduce some trust evaluation mechanism, several issues arise. Imagine for instance that we replace Erin with Eve, which Alice, despite being her friend, consider malicious. Since Alice does not trust Eve, she does not want to use her in the computation of valid signatures to authenticate Bob's public key, but, even more importantly, she does not want her friends to use Eve's SMP verifications for their WoT. For this reason, Alice in her friend list will advertise that to verify Bob's public key she used Dorothy at distance 1 and Henry at distance 4, but not Eve. However Eve can inspect Alice's friend list and would see that she used a rather distant user, Henry, but not Eve which would be a single step away, and can therefore understand that Alice marked her as an untrusted friend. This is an unintended leakage of information which could lead to problems for Alice or, even worse, lead her to mark all her friends as trusted to avoid possible retaliations.

The leakage obviously does not concern only explicit distrust, but also all the possible degrees of trust one could be willing to set for a friend, such as «fully trusted» or «marginally trusted», as it happens in GnuPG. GnuPG does not suffer from this problem since the user can observe the whole WoT and assigns trust evaluations privately, which is not possible in our scenario.

This problem is present also in the scenario of Figure 3.3a on page 53, but its impact is negligible since Alice, for unspecified reasons, could decide to authenticate Bob through SMP even if she has apparently enough confirmations of the correctness of his public key. The leakage of information relies in the choice of including or omitting a particular friend in the advertised list of confirmations for a friend's public key. In general it is hard to solve this problem keeping the WoT private and without leaking information about trust. A possible workaround might consist in adding untrusted friends to a dedicated group where the WoT is explicitly disabled, but identified as an «acquaintances» group.

Due to this difficulties we opted for the usage of WoT only in case at least 3 friends have directly verified Bob's public key with SMP and leave for future works to further investigate trust chains and ways to exploit them to use sophisticated metrics for trust computation (such as AppleSeed [106]) without leaking sensitive information.

Algorithm 3.2 The Socialist Millionaire Protocol

1. The initiator \hat{A} does the following:
 - (a) Pick $u_2, u_3 \in_R [1, q - 1]$
 - (b) Compute, for a fixed g_1 , $U_2 = g_1^{u_2}$ and $U_3 = g_1^{u_3}$
 - (c) Choose a question C
 - (d) Add $\text{Enc}_{K_2}(C, U_2, U_3)$ to message 1
2. Upon receiving and decrypting message 1, \hat{B} does the following:
 - (a) Validate U_2 and U_3
 - (b) Pick $v_2, v_3 \in_R [1, q - 1]$
 - (c) Compute $V_2 = g_1^{v_2}$ and $V_3 = g_1^{v_3}$
 - (d) Derive two keys: $g_2 = U_2^{v_2}$ and $g_3 = U_3^{v_3}$
 - (e) Answer question C with $R_{\hat{B}}$
 - (f) Compute $K_{\hat{B}} = H(R_{\hat{B}}(C))$
 - (g) Compute $o = \text{MAC}_{K_{\hat{B}}}(\hat{A}, \hat{B}, t_A, t_B)$
 - (h) Pick a $v \in_R [1, q - 1]$
 - (i) Compute $(P_v, Q_v) = (g_3^v, g_1^v g_2^v)$
 - (j) Add $\text{Enc}_{K_2}(V_2, V_3, P_v, Q_v)$ to message 2
3. Upon receiving and decrypting message 2, \hat{A} does the following:
 - (a) Validate V_2 and V_3
 - (b) Derive $g_2 = V_2^{u_2}$ and $g_3 = V_3^{u_3}$
 - (c) Answer question C with $R_{\hat{A}}$
 - (d) Compute $K_{\hat{A}} = H(R_{\hat{A}})$
 - (e) Compute $p = \text{MAC}_{K_{\hat{A}}}(\hat{A}, \hat{B}, t_A, t_B)$
 - (f) Pick a $u \in_R [1, q - 1]$
 - (g) Compute $(P_u, Q_u) = (g_3^u, g_1^u g_2^u)$
 - (h) Compute $R_u = \left(\frac{Q_u}{Q_v}\right)^{u_3}$
 - (i) Add $\text{Enc}_{K_2}(P_u, Q_u, R_u)$ to message 3
4. Upon receiving and decrypting message 3, \hat{B} does the following:
 - (a) Compute $R_v = \left(\frac{Q_u}{Q_v}\right)^{v_3}$
 - (b) Check that $o = p$ computing $R_v^{v_3} = \frac{P_u}{P_v}$
 - (c) If they match set $W_{\hat{B}} = 1$ otherwise set $W_{\hat{B}} = 0$
 - (d) Add $\text{Enc}_{K_2}(R_v, W_{\hat{B}})$ to message 4
5. Upon receiving and decrypting message 4, \hat{A} does the following:
 - (a) Check that $o = p$ computing $R_v^{u_3} = \frac{P_u}{P_v}$
 - (b) If they match set $W_{\hat{A}} = 1$ otherwise set $W_{\hat{A}} = 0$
 - (c) Add $\text{Enc}_{K_2}(W_{\hat{A}})$ to message 5

Chapter 4

Group management

In this chapter we describe the system used in SNAKE to efficiently manage groups.

In the following we first explore the state of the art, then we describe the solution adopted in SNAKE for group management and we conclude with an analysis of the costs for the proposed solution.

4.1 State of the art

In OSNs the communication between users is not limited to one-to-one messages exchange. The alternate form of communication is represented by many-to-many messages, which, in turn are far more common than one-to-one messages. Apart from private communication between two users, all the other activities performed on a OSN involve sending many-to-many messages. It is therefore crucial to have an efficient way to handle this type of messages.

Messages sent to what we call groups is precisely a form of many-to-many communication. The confidentiality of messages sent to the group is guaranteed through encryption using a symmetric group key. We opted for this method since it is the most efficient way to send a message to multiple recipients. However, there is another aspect that must be taken into account: the management of these groups. Groups are dynamic, new members can join the group and some members can be removed from the group. To ensure the confidentiality of messages when a member joins or leaves the group it is necessary to change the group key, this operation is called *rekeying*.

If a new element joins a group, the group key can simply be individually

sent to him and to the existing group (encrypted with the previous group key) to inform the old members of the change. However after a user leaves the group, the previous group key cannot be used to encrypt the new one, because the removed user knows that key. The naïve approach to this problems is to send the new group key individually to each user still in the group, which requires to communicate this key using one-to-one messages. The messages used to inform the members of the group that the group key is changed are called *rekeying messages*.

The number of times a key has to be encrypted and sent to inform all the users that the group key is changed is called *rekeying cost*. We note that in literature, different definitions of rekeying cost have been given, sometimes it is expressed as the size of the messages that a member of the group has to decode, or as the size of the messages that needs to be encrypted and sent to the group, and other definitions. We chose to express the rekeying cost as the sum of the number of keys that are encrypted as it is a direct measure of the network traffic generated for a rekey operation.

The naïve approach leads to a rekeying cost proportional to the number of users in the group. When dealing with large or dynamic groups this leads to severe scalability problems. Moreover, the update of the key caused by a member's departure cannot be notified within the group but requires sending a several one-to-one messages. Wallner et al. [100] and independently Wong et al. [104] proposed a hierarchical key-tree approach that allows to have a rekeying cost that is logarithmic in the number of users of the group. The system proposed is often called Logical Key Hierarchy, or simply LKH.

4.1.1 Logical Key Hierarchy

In this section we will illustrate in detail how LKH is able to efficiently handle groups evolution.

4.1.1.1 Notation

Since the notation used among different sources is not uniform we will use the following definitions. When talking of B-Trees we will use d to denote the degree of the tree, that is the maximum number of children of a node and also the maximum number of values that can be stored inside leaf nodes. h is the height of the tree, with the root node having height 0 and the height is always

$\lceil \log_d n \rceil$. t is the minimum number of children of a node and is always $\lfloor \frac{d}{2} \rfloor$. n is the number of participants in a group. k_1 is a symmetric encryption key with the id 1. $[k_1]$ is a node of the tree with an associated cryptographic key. $\{k_1\}_{k_2}$ is a message that contains the key k_1 encrypted with the key k_2 .

4.1.1.2 Key graph

LKH employs a key graph that is a B-Tree in which are stored the users of the group, the peculiarity of this tree is that every node has an associated cryptographic key. Figure 4.1 shows an example of tree that contains 7 users. Leaf nodes represent users in the group. Internal nodes represent a set of subgroups of the group, each key inside internal nodes is used to securely communicate with subgroups of the group (i.e. k_{45} can be used to communicate with u_4 and u_5). The root node represents the whole group and its key is used as the group key.

Each user knows only the keys contained in the nodes on the path from the root node to the leaf node that represents him. For example the user u_1 has the keys: k_1 that is only known to him; k_{13} that is known to him, u_2 and u_3 ; k_{17} that, being the group key, is known to all the users. When there is the need to inform all the users that the group key has changed, the keys of internal nodes are used to inform more than one user with a single message, instead of sending individual messages to each user.

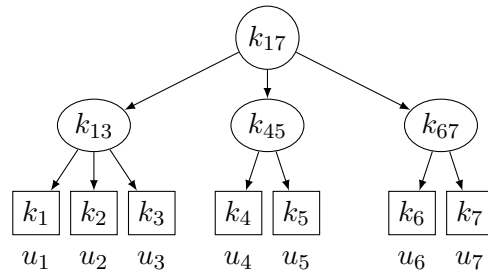


Figure 4.1: LKH: key graph representing a group of users

The tree is updated at every join or departure of a member and the rekeying message is computed by modifying it. In the following we will explain the actions required to insert or remove a member from the tree. The following examples use a tree with degree $d = 3$ and $t = 1$.

4.1.1.3 Insertion

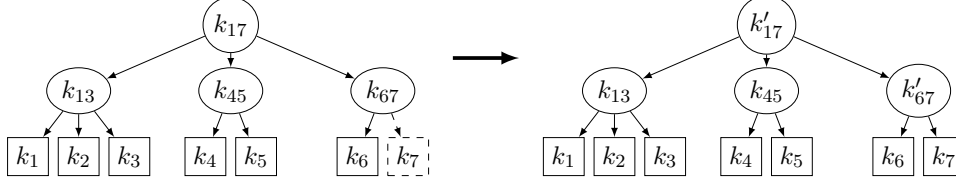


Figure 4.2: LKH: a new member joins the group

When a new user is added to the group a new leaf node is created. Figure 4.2 shows an example of user insertion: the node $[k_7]$ is created, it represents the user u_7 , which is added as a child of $[k_{67}]$. After the insertion we have to change all the keys of nodes on the path from the root node to the leaf node of u_7 , which are k_{17} and k_{67} . For a secure distribution, the new key k'_{17} can be encrypted with its previous value k_{16} because it is not known to the new user, the same thing applies to the key k'_{67} . The new user needs to know all the keys contained in the nodes from the root node to the leaf node associated to him, which are k'_{17} , k'_{67} . Those keys are encrypted with k_7 that is only known to u_7 . The rekeying message for the insertion of u_7 represented in Figure 4.2 is $\{k'_{17}\}_{k_{17}}$, $\{k'_{67}\}_{k_{67}}$ for the members of the group, $\{k'_{17}, k'_{67}\}_{k_7}$ for the new user. Therefore, the insertion of a new member requires to change and encrypt h keys for the previous members of the group, and other h keys are encrypted and sent to the new user, leading to a the best-case rekeying cost for inserting a new user of $2h$.

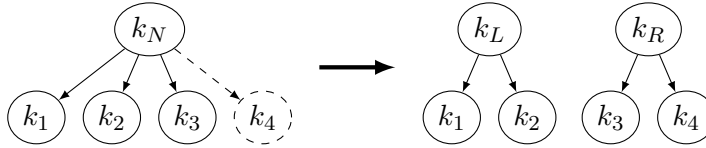


Figure 4.3: Node $[k_N]$ is splitted in nodes $[k_L]$ and $[k_R]$

An insertion can bring a non-leaf node to have more than d children, re-balancing a B-Tree after insertions is achieved by node splitting. Figure 4.3 shows an example of node splitting. Since the maximum number of children is 3, the node $[k_N]$ needs to be split in two nodes $[k_L]$ and $[k_R]$. For a secure distribution, the key k_L must be encrypted using the keys of its child nodes

(k_1 and k_2), the same applies for k_R . The rekeying message for the split operation represented in Figure 4.3 is $\{k_L\}_{k_1}, \{k_L\}_{k_2}, \{k_R\}_{k_3}, \{k_R\}_{k_4}$. Splits occurs when a node has $d + 1$ children, therefore the generic rekeying cost for this operation is $d + 1$. A split may be propagated to the root node, splitting up to h nodes, leading to the worst-case rekeying cost for inserting a new user of $h(d + 1) + h$.

4.1.1.4 Removal

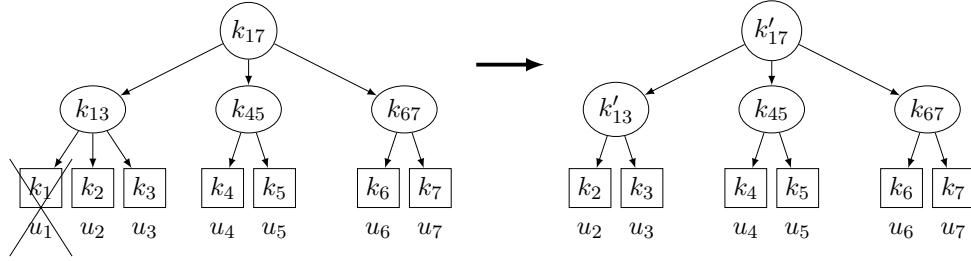


Figure 4.4: LKH: a member is removed from the group

When a member is removed from the group the leaf node associated to him is removed from the tree. After the removal we have to change all the node keys on the path from the root node to the leaf node of u_1 , which are k_{17} and k_{13} . The rekeying process works bottom-up, from the parent node of $[u_1]$ to the root node. k'_{13} is encrypted with the keys of the child nodes of $[k'_{13}]$ (k_2 and k_3), then the same thing applies to k'_{17} . The rekeying message for removing u_1 as represented in Figure 4.4 is $\{k'_{13}\}_{k_2}, \{k'_{13}\}_{k_3}, \{k'_{17}\}_{k'_{13}}, \{k'_{17}\}_{k_{45}}, \{k'_{17}\}_{k_{67}}$. Since each node can have a number of children that varies from t to d (2 to d for the root node) the rekeying cost is variable. However, the best case occurs when all the nodes have t children and the root has 2 children, which leads to a best case rekeying cost of $d - 1 + th$.

A removal can bring a non-leaf node to have less than t children. After a deletion a rebalanced B-Tree is achieved using two different methods: redistribution or merge.

If the node to rebalance has a sibling with more than t children it is performed a redistribution as showed in Figure 4.5. For a secure distribution, the key k'_L must be encrypted using the keys of its child nodes (k_1 and k_2), the same thing applies for k'_R . The rekeying message for the redistribute operation

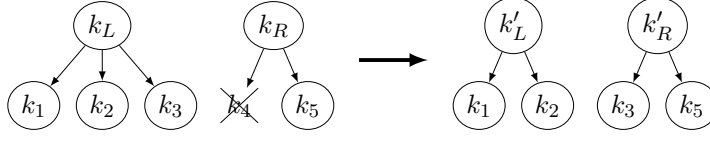


Figure 4.5: An element of $[k_L]$ node is redistributed in node $[k_R]$

as represented in Figure 4.5 is $\{k'_L\}_{k_1}, \{k'_L\}_{k_2}, \{k'_R\}_{k_3}, \{k'_R\}_{k_4}$. A redistribute operation occurs when the node to balance has $t - 1$ children and there is a sibling with at least $t + 1$ elements. The keys of the node to balance and of his sibling have to be changed and are encrypted with the keys of the child nodes. Therefore, the rekeying cost for a redistribute operation is between $2t$ and $t - 1 + d$. The redistribution is an operation that does not propagate to the parent node where it is applied, once done the tree is balanced.

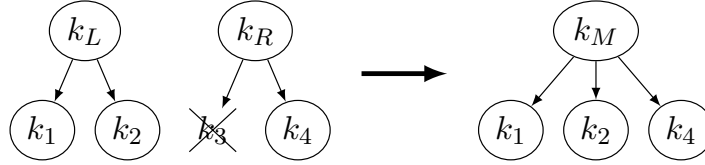


Figure 4.6: Nodes $[k_L]$ and $[k_R]$ are merged into one node $[k_M]$

If the node to rebalance has no siblings with more than t children, it is merged into one of its siblings as showed in Figure 4.6 on page 64. A new node $[k_M]$ is created, where all the children of $[k_L]$ and $[k_R]$ are attached. For a secure distribution, the key k_M must be encrypted with the keys of the child nodes of the node where the removal was performed. However, since the children of node $[k_L]$ have not been involved in the removal, it is possible to use k_M to perform the encryption. The resulting rekeying message for the merge operation, as represented in Figure 4.6, is $\{k_M\}_{k_L}, \{k_M\}_{k_4}$. A merge operation occurs when the node to balance has $t - 1$ child and the node merged has t children, therefore the generic rekeying cost for this operation is $t - 1$. The merge operation could spread up to the root node, merging h nodes, leading to a worst-case rekeying cost of $d - 1 + th$.

4.1.1.5 Rekeying costs

We showed how insertions and a deletions works for LKH, Table 4.1 summarizes the costs associated to the operations that are required to maintain a B-Tree balanced.

Table 4.1: Rekeying costs for basic operations

	Split	Merge	Redistribute	
			min	max
Cost	$d + 1$	t	$2t$	$t - 1 + d$

As the comparison in Table 4.2 shows, LKH allows to have much lower rekeying costs compared to the presented naïve solution.

4.1.2 Improvements over LKH

In the last decade there has been a lot of work aiming to improve the LKH system. An important result, obtained by Snoeyink et al. [96], has proved that using this key-based systems the lower bound for the rekeying cost is $\theta(\log n)$.

Canetti et al. [18] proposed an improvement over the LKH scheme that allows to have a rekeying cost at insertion of $h + 1$ instead of $2h$. h keys still need to be updated in the tree, but it is sufficient to send to the new user only a single key instead of h . Using the original LKH scheme, when a key is changed, the new one is randomly chosen. The improvement proposed in [18] consists in deriving the h new keys using a PRNG starting from a random value generated for every insertion.

Another improvement that is also able to halve the rekeying cost for deletion has been presented by Sherman et al. [94]. They use an algorithm called

Table 4.2: Rekeying costs for the insertion of a new member

	Insert		Delete	
	best case	worst case	best case	worst case
Naïve	1	1	n	n
LKH	$2h$	$(d + 1)h + h$	$t(h - 1) + 2$	$d - 1 + th$

One-Way Function Tree (OFT) that works on binary trees. Each node of the tree has, in addition to a key, a secret value x . Only leaf nodes have an actual x value, while for all the internal nodes x is derived through a one-way function that uses the secret value of the left and right children of the node. This way, the OFT offers a rekeying cost that is the half of LKH for both insert and remove operations.

However subsequent works showed that OFT is vulnerable to a series attack where users that collude are able to continue to read the group conversations even if they were removed from the group. Several variants to the original OFT have been proposed to mitigate or eliminate this kind of problems but with an higher cost with respect to the proposed solution. In [105] the main issues are summarized.

There are a lot of other minor modifications over LKH, most of them oriented to adapt the original solution to very specific needs (e.g. pay per view systems) or application fields (e.g wireless sensor network, multimedia applications...).

All these improvements have one thing in common, they did not change the data structure used to represent the key graph: it is always a B-Tree. Maintaining the tree balanced requires a series of operations that increases the rekeying cost for insertions or removals, using a different data structure or modifying the existing one may lead to better results. There are some works that have considered this approach showing interesting results.

Rodeh et al. [88] proposed a system using AVL trees rather than B-Trees, it is an interesting solution to have a distributed tree among the users instead of a centralized solution, but it does not reduce the rekeying cost with respect to LKH.

Goshi et al. [42] carried on an extended analysis of some alternatives data structure. They proposed three new algorithms for the maintenance of key trees based on 2-3 tree and 2-3-4 trees. They also empirically evaluated the worst-case rekeying costs, showing that they are able to achieve better results than standard B-Trees.

A novel solution presented by Lu et. al [64] introduces a new type of tree called NSBHO (Non-Split Balancing High-Order). We describe in detail this new approach since it contains some important changes that are worth to be explained.

4.1.3 NSBHO

NSBHO is a tree similar to a B-Tree where split operations are not performed. Avoiding this type of operations allows to achieve a lower rekeying cost for the insertion of a member. The idea behind this choice is to perform insertions by attaching leaf nodes to any internal node that has enough space to contain it. Leaf nodes are all placed at the same depth to keep the tree balanced. This is done by using a chain of internal nodes where the leaf node is attached. Figure 4.7 shows an example of this idea applied to the insertion of a member in a group. The tree has a degree $d = 3$ and $t = 1$.

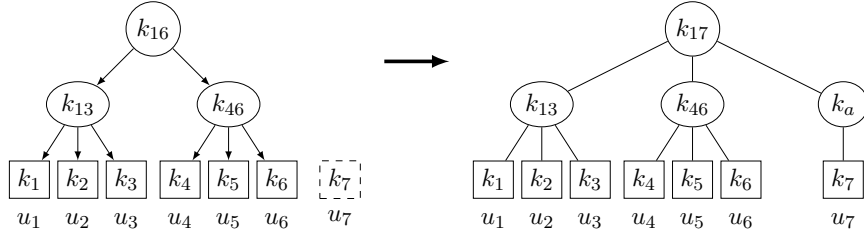


Figure 4.7: NSBHO: a new member joins the group

The only internal non-full node is $[k_{17}]$ where the chain of nodes $[k_a]$, $[k_7]$ is attached. The internal node $[k_a]$ is used to place the leaf node at the correct depth that maintains balanced the tree. When there are no non-full internal nodes the tree is full, and the insertion of a new member is performed by increasing the height of the tree. A new root node is created on top of the old tree and the leaf node of the new user. An example of insertion that causes the root to increase its height is showed in Figure 4.8.

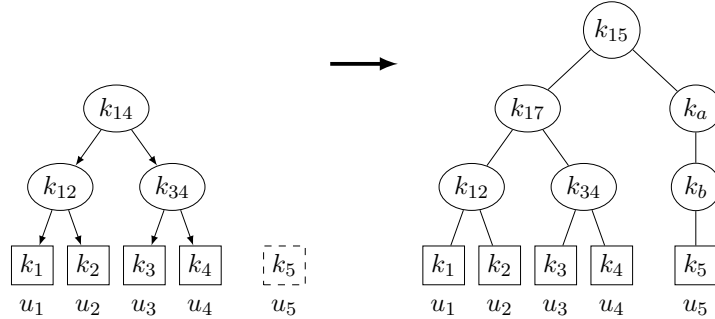


Figure 4.8: NSBHO: a new member joins a group whose associated tree is full

The internal nodes that are created when the insertion of the leaf node is not performed in the penultimate level of the tree are part of a path called special path (SP). The special path is defined as «a sequence of internal nodes, (z_0, z_1, \dots, z_k) , where z_i is an ancestor of z_{i+1} for $0 < i < k$, z_i has at least one child and at most $t - 1$ children for $0 \leq i \leq k$, and z_0 is not the root» [64]. It is a path of the tree where the nodes do not respect the properties of a standard B-Tree. NSBHO trees can have at most one SP.

The difference between a standard B-Tree and the NSBHO tree is that the latter is not a search tree. Inserting the elements in the first free space does not preserve the order of the leaf nodes that is needed to use the tree as a search tree.

For the insertion of a new node we need two separate lists containing the references to the non-full internal nodes and to the nodes in the SP. These two lists will be used to choose the internal node where to attach the external node.

Removal of elements is performed as in LKH with a few small changes to handle nodes that are in the SP. Since NSBHO is not a search tree, removals are performed directly from the leaf nodes, which requires to store the references to all the leaf nodes in another dedicated data structure.

A comparison of the rekeying costs of the two systems is showed in Table 4.3.

Table 4.3: Comparison of rekeying cost between LKH and NSBHO

	Insert		Delete	
	best case	worst case	best case	worst case
LKH	$2h$	$(d + 1)h + h$	$t(h - 1) + 2$	$d - 1 + th$
NSBHO	$h + 1$	$2h$	1	$d - 1 + th$

4.2 Proposed approach

In SNAKE, we decided to employ key graphs for group management, since this approach allows us to handle rekey within the group in an efficient way.

4.2.1 Key graph approach

In contrast with the various solutions presented, we decided not to perform a rekey when a new member joins the group. This is an important design choice justified by the nature of our groups. In OSNs when a new member joins a group or becomes a friend of another user, he has full access to the message history of that group. Therefore, it is useless to perform a rekeying when a new member joins, since he will have access to the previous conversations anyways.

Keeping this choice in mind, we decided to use NSBHO to handle the group key graph. By using this system it is possible to have the lowest possible rekeying cost for the join of a member. Although we decided not to change the group key upon member insertion, the rebalancing of the tree may require to split some nodes. With LKH those splits still need a rekeying message, while with NSBHO the problem is completely bypassed as splits are completely avoided. The only case in which a message is required for an insertion in a NSBHO tree is when the tree size increases. However in that case the rekeying cost is unitary.

Users save, for each group they are in, the current group key along with the h other auxiliary keys used to read rekeying messages.

4.2.2 Join or leave a group and administrative privileges

Only administrators of a group have the authority to add or remove group participants. The removal of a participant from the group is handled completely within the group: the administrator updates the administrative data of the group and then sends the rekeying message to the group. When the removed user reads the rekeying message, he can only understand that he has been removed, but he cannot decrypt any of the new keys contained in the message.

The insertion of a member into the group cannot be managed exclusively within the group. It is necessary to have a secure way to communicate between an administrator and a member that wants to join a group to exchange three messages: a request of the user to join a group and a positive answer of the administrator to the request that contains the group key together with the auxiliary keys or a negative answer to the request.

In the beginning the creator of the group is the only administrator, but this privilege can be extended to other members. The information needed to

become a group administrator are: the symmetric key used to encrypted the `adminData` field of the group descriptor and the private key used to sign the group descriptor.

4.3 Implementation details

The administrative data of the group is stored in the `adminData` field of the group descriptor. When requesting the group descriptor this field is not fetched because it contains only information needed by administrators, members and subscribers are not able to decrypt it. A separate network request exists to get the group descriptor with the `adminData` field, which is used by administrator when they have to modify the group.

`adminData` contains a single property named `serializedTree` that represents the tree used to manage the group evolution.

4.3.1 Tree serialization and deserialization

When the tree is loaded into memory, it is represented as nested JavaScript object. This object contains some redundant information that cause circular dependencies. Each node has a reference to its parent node and to its child nodes, the parent node causes a circular reference. The other circular dependencies are caused by the two arrays that contain the list of non-full nodes and the nodes of the SP. Moreover, the object in which are stored references to all the leaf nodes causes circular references. These elements that cause those circular dependencies are however extremely useful when performing an insertion or a deletion.

When the tree is serialized all of these dependencies are removed: in each node the reference to the parent node and the two arrays mentioned above and the object with the references to non-leaf nodes are removed. This is not a problem since all the information can be reconstructed when the object is deserialized.

The deserialization process restores the tree and all the redundant information that are used to perform insertion and deletions. Since this tree enrichment operation takes place in a single pass, while the JSON object is parsed, the additional overhead is negligible.

4.3.2 Rekeying messages

Each node in the tree has a key and a unique ID that is used to identify different auxiliary keys. The rekeying message consists in a series of key update directives composed of 3 fields:

encryptedKey Contains the new key encrypted with another key.

keyID Contains the ID the key contained in the **encryptedKey** field.

encryptedWith The ID of the key that must be used to decrypt the new key.

We used AES-CBC to encrypt the keys rather than AES-GCM as for these keys there was no need for an additional integrity check since they are sent in an already authenticated post. The IV used to encrypt all the keys of the same rekeying message is the same.

The rekeying directives and the IV are included in a post that is sent to the group. This post also contains information about who is the user that has been inserted or removed and which administrator performed the operation.

Posts containing a rekeying must be processed only if sent by a group administrator.

4.4 Cost-benefit analysis and engineering tradeoffs

There is a critical parameter which must be properly selected in an implementation of our proposed system of group management: the degree d of the tree. In fact, it affects three different aspects:

Size of the tree Intended as the number of nodes (or keys) stored in the tree.

Size of rekeying messages This size depends directly on the degree of the tree.

Size of user keys The number of keys that a user has to store, which corresponds to the height h of the tree.

A first consideration is that, since we set $t = \lfloor \frac{d}{2} \rfloor$, it is better to choose an odd degree instead of an even. The reason for this is that the tree requires less maintenance (merge and redistribution). For instance, consider the case of a tree with degree $d = 5$ having $t = \lfloor \frac{d}{2} \rfloor = 2$ leads to nodes that have a lower probability to be one child short rather than having $t = \lceil \frac{d}{2} \rceil = 3$.

The degree has to be chosen taking into account the particular nature of the group, i.e. in a group composed by thousands of users is better to have an high degree since it minimizes the size of the tree, while with a very dynamic group is better to have a low degree that minimizes the dimension of rekeying messages. There is not an optimal value valid in every scenario, therefore we analyzed the problem to find a degree that best fits our application.

The optimal degree for a group is the one minimizing the network traffic generated by its management. The following formula calculates the amount of generated traffic in terms of keys sent over the network.

$$F(i, r, d) = \sum_{k=1}^{\text{size}(i)} [i_k \cdot \text{insertRekeyCost}(i_k, d) + \text{treeSize}(i_k, d) + \text{treeSize}(i_k + 1, d)] + \sum_{k=1}^{\text{size}(r)} [r_k \cdot \text{removeRekeyCost}(r_k, d) + \text{treeSize}(r_k, d) + \text{treeSize}(r_k - 1, d)]$$

where:

i is the list of tree sizes in which an insertion is performed

r is the list of tree sizes in which a deletion is performed

insertRekeyCost (x, d) number of keys in the rekeying message for insertion of a new member in a tree of degree d with x users

treeSize (x, d) number of keys in a tree of degree d with x users

removeRekeyCost (x, d) number of keys in the rekeying message for the deletion of an user in a tree of degree d with x users

i and r are a list of tree sizes because we have to take into account the group evolution. If we consider a group of 5 user the cost to build its tree is the sum of the insertion costs in a tree with 0, 1, 2, 3, 4 users.

The first summation is for all the insertions performed in the group: the insert rekeying cost is multiplied by the number of users in the group, since every member of the group has to fetch the rekeying message. The second summation is for all the deletions performed in the group.

We need three functions (insertRekeyCost, removeRekeyCost and treeSize) that, given a tree with a particular degree d and x users, are respectively able

to compute the size of the associated tree, the rekeying cost to add a new member and the rekeying cost to remove a member.

There is not a formula that counts the number of nodes in a tree, there are only upper and lower bounds. Theoretical rekeying costs cannot be used to estimate real rekeying costs for a member removal because they provide only upper and lower bounds. For instance, NSBHO has a minimum rekeying cost for the deletion of 1, however this case is very rare and occurs only when the member to remove was previously inserted when the tree was full. The exact function to compute rekeying cost at insertion gives 1 when the tree is full and 0 otherwise:

$$\text{insertRekeyCost}(x, d) = \begin{cases} 1 & \text{when } \lfloor \log_d x \rfloor = d^i, \quad i \in [1, +\infty] \\ 0 & \text{otherwise} \end{cases}$$

To determine a practical approximation of the exact function for the size of the tree and the rekeying cost for removal we run two simulations to measure those costs. We created a series of trees with a number of users ranging from 0 to 1000 and, on those trees, we measured: the size of the tree and the cost for user removal. The trees were created using random insertion and deletion and the simulations were averaged over 1000 runs. The results of the simulations are shown in Figure 4.9 on page 74 and Figure 4.10 on page 74.

Looking at the graphs it is clear that the rekeying cost at removal and the tree size are respectively logarithmic and linear in function of the group size. With the measured costs it is possible to derive two functions fitting the obtained costs. The mathematical method we used to derive the two functions was the well known Levenberg–Marquardt algorithm [71] for non-linear regression. We used the implementation provided by the function `leasqr` [95] in Octave with the following models for the regression:

$$\begin{aligned} \text{removeRekeyCost}(x, d) &= p_1 \cdot \log_d(x) + p_2 \\ \text{treeSize}(x, d) &= p_1 \cdot x + p_2 \end{aligned}$$

Table 4.4 on page 75 shows the result of the regression. An important output given by the Levenberg–Marquardt algorithm is R^2 , the coefficient of determination, which is an index for how well the model fits actual data points. The coefficient ranges from 0 to 1, where 1 is a perfect fit and 0 means

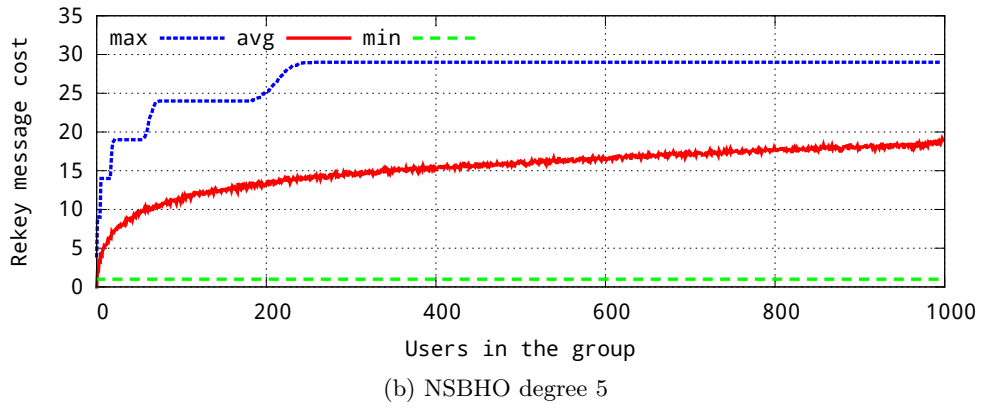
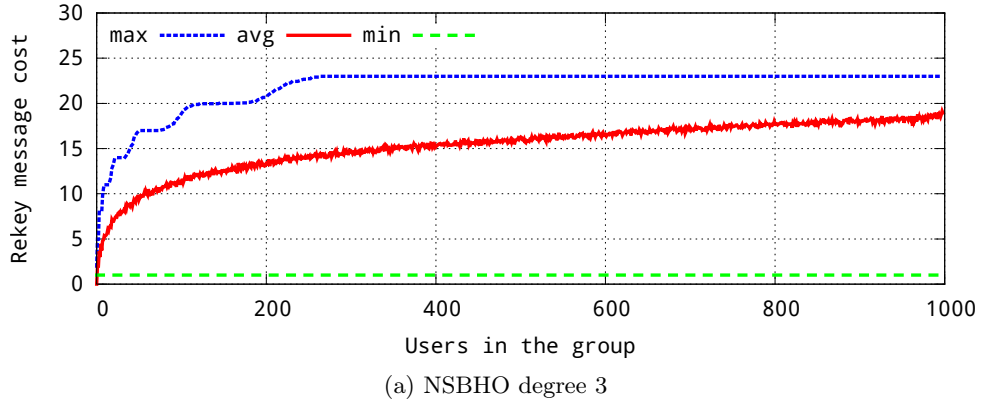
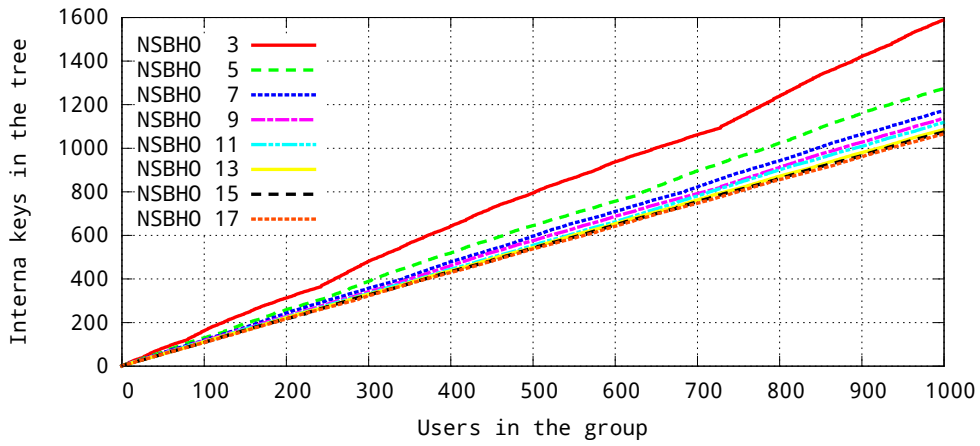


Figure 4.9: NSBHO real rekey cost for deletion



an absolute lack of correlation between the data and the used model. Looking at the determination coefficient for our data it is clear that a linear function almost perfectly models the tree size. Also the logarithmic function fits well the rekeying costs at removal, but the fit quality slightly decreases with higher degrees.

Table 4.4: Results of regression to determine real rekeying cost at removal and tree size for NSBHO

Order	removeRekeyCost			treeSize		
	p_1	p_2	R^2	p_1	p_2	R^2
3	3.24	-2.19	0.99322	1.55	3.87	0.99875
5	5.78	-5.47	0.95572	1.27	1.74	0.99975
7	8.23	-8.17	0.90851	1.17	3.84	0.99991
9	11.25	-11.84	0.85794	1.13	1.44	0.99989
11	14.56	-15.84	0.82435	1.12	2.80	0.99986
13	15.90	-16.26	0.77407	1.08	2.38	0.99999
15	18.00	-17.93	0.73259	1.07	2.22	0.99998
17	20.47	-20.06	0.70424	1.06	2.02	0.99998

With an expression for insertRekeyCost, removeRekeyCost and treeSize is possible to estimate a value of d minimizing our objective function F :

$$\arg \min_d = F(i, r, d)$$

We considered a series of 1000 sequential insertions ($i = [0, 1, 2, \dots, 999]$) and a different number of removals (0, 10, 20 50, 100, 150, 500 and 1000) to simulate various types of groups. The simulation was performed 1000 times, for each iteration the values of r were chosen randomly to average the result for the costs.

Figure 4.11 on page 76 is the plot of function F for a set of different degrees, the results are not the absolute rekeying costs, the minimum value of each set has been normalized to 0 and the maximum to 1. The graph only reports three different ratios of removals over insertions for clarity. It can be noticed that for a group without removals it is better to have a tree with an high degree, while for a very dynamic group is better to have the lowest possible degree. Considering a percentage of removals over insertion of 5 – 20%, which is the case of the groups that we are considering, the advantage of using an higher

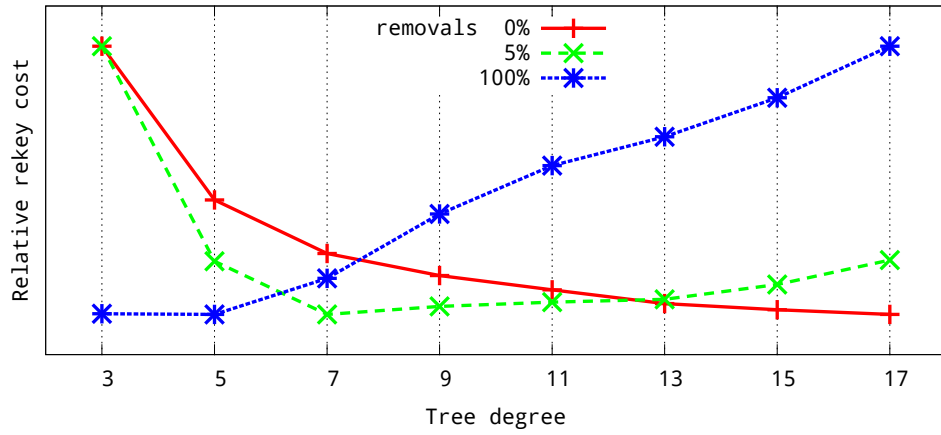


Figure 4.11: NSBHO degrees removals statistics

degree vanishes around values greater than 9. Therefore our choice was to use 7 as for the degree of our tree.

Chapter 5

Anonymity of the stored data

In this chapter we briefly describe how we protect metadata of communications taking place over SNAKE and in particular how to keep the social graph secret.

In the following we first explain how to protect communication metadata by masking the edges of the social graph, then we present some countermeasures to prevent a remote user from obtaining a complete dump of the database. We will not discuss the privacy issues of the WoT table since its privacy has already been considered in Section 3.4.

5.1 Choice of the scenario

We consider the HONEST-SERVER scenario presented in Subsection 2.2.2, which, in brief, considers a collaborating and honest storage server. We also thoughtfully considered if it was possible to anonymize the social graph to the eyes of an untrusted server. In particular we explored ways to make private queries to a database, analyzing solutions ranging from *oblivious RAM* (introduced in [41]) to more recent solutions such as the *shuffle index* proposed by De Capitani di Vimercati et al. [27]. While these approaches are getting progressively more viable, they still generate a too large network overhead to use them in a highly interactive system such as SNAKE. As already highlighted in the Frientegrity paper [33], without these radical approaches any other mitigation would be ineffective. In fact, even considering the web browser as completely untraceable (for instance using the Tor Browser Bundle [85]), its user would be identifiable through his behavior: the simplest attack would be to fingerprint

him through the list of requests he does upon login.

While the ideal situation would be to be able to hide metadata from the storage server too, protecting it with data *at rest* results to be very useful in case the hardware is seized, system violations or questionable requests of information by law enforcement (see the Lavabit case [1]).

Moreover this approach does not only protect the users, but also the storage provider itself, since it cannot be held responsible for what the users share among them or with whom they communicate, since it has no technical mean to get access to it. This is also the approach followed by the re-born Megaupload website: Mega.co.nz [63].

5.2 Preserving social graph secrecy

We carefully designed SNAKE to avoid as much as possible the leakage of information about the social graph on the long-term storage. In particular, we asked ourselves what were the essential fields we had to keep in a form intelligible by the storage server to keep the service working without mayor performance drops. In practice, those fields are primary indexes, i.e. fields used by the client and the server to uniquely identify each record.

Let us consider the **User**, **Profile** and **Group** tables: we cannot eliminate their primary keys, but there is no reason for which they should leak the order in which the records have been inserted, as it happens if progressive integers are used as primary keys. We can store them as a random identifiers, large enough to be considered unique (128 bits in our implementation).

For the **Message** table the situation is not as easy, since it must support range queries, for instance to fetch all the messages posted to a group after a certain point in time, usually the last visit of the user.

A first solution we found consisted in storing each message along with a timestamp, the identifier of the recipient and a secret identifier for the sender known only to the recipient. Instead of a timestamp, a progressive integer could be used, but, assuming enough activity on the table, they have similar properties.

For this configuration, an attacker could detect from the pattern of the timestamps and the identifiers of the recipients that a live conversation, or friendship handshake, took place. In Figure 5.1 on page 79 we show how messages to establish a friendship look like in the described situation. If both

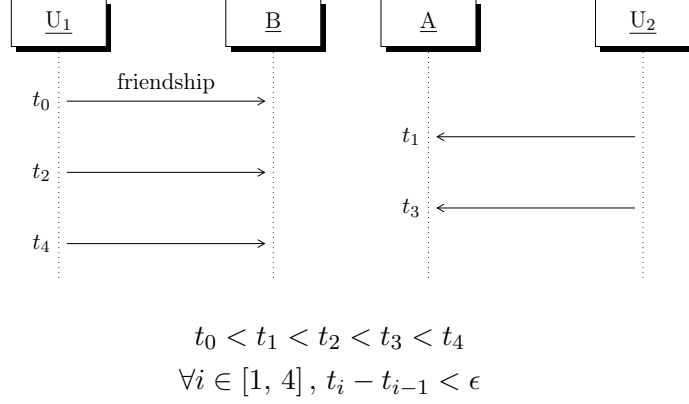


Figure 5.1: How a friendship establishment looks on the database if timestamps or progressive numbers are used to identify private messages. t_i indicates an instant in time or a progressive number. U_1 and A are aliases for Alice, while B and U_2 are Bob's aliases.

users are online, the exchange of messages will take place in a short time span. Therefore an attacker, with the help of the size of messages, can easily tell that a friendship handshake between Alice and Bob was going on.

For this reason we decided to use as primary key a completely random identifier we call **address**, along with a progressive integer, called **index**, which grows independently for each address. Doing so, it is not possible to determine the order of messages addressed to different users and perform the described attack.

In summary, each message is sent to an address, which corresponds to a group or a user, and a progressive **index** is assigned to it. When the associated user, or the members of the associated group, want to check if new messages are available, they will make a request to the storage server asking for records matching that address and with an **index** value greater than the last they saw. In the following we detail how the three main type of entities stored in the **Message** table are handled.

Private messages Sender and recipient identifiers are not stored explicitly, we use an address which identifies a unidirectional communication channel between two users. This means that when Alice sends a message to Bob she sets as recipient an address they previously agreed upon. When Bob checks if he has new private messages, he asks the server all recent

messages addressed to one of the addresses he agreed with his friends. Bob is able to understand which key he has to use to decrypt each message from the used recipient address.

Friendship establishment messages The agreement on the address to use when exchanging private messages is reached during friendship establishment. For this reason the first message of the handshake cannot use this address. Indeed the recipient address for friendship requests has to be public and available to everyone: it is stored in the `friendshipAddress` public field, as shown in Figure A.2 on page 134. If Alice wants to establish a friendship with Bob, she has to send a message to that address, embedding her own identifier encrypted as shown in Subsection 3.3.1. She also sends the address Bob has to use to write her in future, that is the $A \rightarrow B$ address. In the answer Bob will inform Alice of the $B \rightarrow A$ address. To further improve privacy, these address can be changed from time to time.

Posts to groups Posts to groups work in way similar to private messages: each group has an address that members use as recipient for their posts. A new member gets to know this address when joins the group. To avoid that a removed user is still able to monitor group activity, upon rekeying the group address is changed and communicated to all participants, except the member just excluded.

As shown in Figure A.1 on page 133, the result of these techniques are the suppression of almost all the candidate join paths, with the exception of: `User.publicProfile` \rightarrow `Profile` and `User.friendshipAddress` \rightarrow `Message`. These two paths allow an attacker looking at the database at rest to easily collect some data, but they do not leak much information, save for what the user has explicitly decided to make public and the amount of not yet handled friendship requests the user received.

So far we tried to design the system to store as little unencrypted data as possible. If we take this reasoning to the extreme consequences, we could even let the database be completely public and completely accessible by everyone. However, as an additional safeguard against vulnerabilities due to implemen-

tation bugs, we designed SNAKE to prevent a standard user from obtaining a complete dump of the database.

From this point of view, the usage of long and random identifiers as primary keys, instead of progressive numbers, offers an additional benefit: it prevents enumeration of the records in the table. For the same reason we also allow only exact-match queries on username for the **User** table.

Chapter 6

User experience

In this chapter we discuss some choices that we made for the implementation of the user interface of SNAKE. We also present the significant differences between our OSN and common OSNs that influence the user experience.

6.1 Design choices

The development of each part of the user interface of SNAKE had a common goal: hide all the underlying technicalities to the user. Our system is complex, performs several cryptographic operations to read and send contents. We could inform the user of every action, showing him that everything is actually encrypted and signed, but we deemed this would needlessly increase the complexity of the user experience. We wanted to show that it is possible to realize a privacy-oriented system that offers the same functionality of a classic OSN with minimal impact on the user experience. Nonetheless, there are some noticeable differences, in the following we will comment the main ones.

6.1.1 Look and feel

A fundamental aspect of any modern website is the so called «look and feel». It affects the way in which the user perceives contents and what he feels just by looking at the website. We are talking about some visual aspects like layout, typography, colors and other aspects of critical importance to obtain an effective result.

We then chose to use Bootstrap [78] framework to create the front-end of our application. Bootstrap is popular framework which provides a well

established library for layouts, forms, buttons and so on. This framework allows us to provide the user with a very familiar interface and to simplify the development of the web interface on our side.

A particularly interesting feature of Bootstrap is that has been designed to provide a Responsive Web Design (RWD). RWD is an approach to web design that allows to have an optimal view experience across a wide range of devices from mobile phones to desktop computer.

6.1.2 Transparent signing and encryption

The encryption is completely transparent to the user. If he has access to the key required to decrypt a content it is simply visualized. If, instead, the user cannot visualize something the client informs him that he does not have the required permission to view that particular content. This behavior is enforced by our client. It is possible to show the public properties of the requested content but this information would only further confuse the user.

Moreover, signature verification is completely transparent. If a signature is valid the message is displayed to the user regularly. If it is not, we still show the content but visually mark it to let the user know that the author of the content can not be verified. Figure 6.1 shows an example of a message whose signature is not valid.

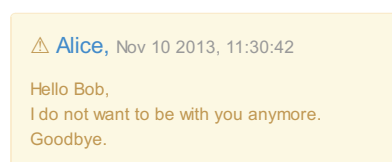


Figure 6.1: Example of a forged message

The idea behind this design is to only show errors to the user. He knows that SNAKE is a OSN where the main focus is on the privacy of the user, but it is not necessary to remind him every time of these properties.

6.1.3 Multiple profiles

Our system offers the possibility to have multiple profiles. A profile is an identity that the user can use on the OSN. This idea is not new to OSNs, Google+ first introduced the concept of circles, then Facebook implemented a

My profiles

Bob	▼	Remove	Edit
Johnny Walker	▼ ▲	Remove	Edit
John Smith	▲	Remove	Edit

Figure 6.2: List of user's profiles

very similar system for dividing friends in Lists. A user can therefore recognize in our «profiles» a similarity with circles and lists and will not incur in major difficulties in using them.

There is a significant difference for the public profile. As described in Chapter 2, its purpose is exclusively to provide some basic information about a user that can help other users to find him. The public profile can be seen as the starting point for a friendship, it should not be used to publish personal information. To stress this fact we removed the possibility to have a group of friends associated to the public profile.

Another minor difference is that each profile has an associated priority, this feature is used to select the profile to show when more than one profile of an user are available. While this priority is stored as an integer number, we chose to let the user select the profiles priority only by ordering them in a list. Figure 6.2 shows what the users see: the first has the highest priority while the last has the lowest priority.

6.1.4 Group management

As described in Chapter 4, we use groups both for group of friends and generic groups. The user does not see that the friends associated to a profile are handled through a group. For this particular group the only administrator is the owner of the associated profile and administrative privileges cannot be granted to someone else. Also, for this groups it is not possible to add members directly as for generic groups. Group of friends are transparently managed in case of friendship establishment or revocation.

As for generic groups, management is completely transparent to subscribers, members and administrators. We do not expose any of the internals of this system to the user, it would be only counterproductive since it would require a knowledge about the technical details of the underlying structure to properly

understand and therefore interact with them.

6.1.5 Friendship

Friendship establishment is performed as in other OSNs: there is a dedicated button on the public profile of the user, a user clicks when he wants to add him as a friend. However friendship establishment requires public key authentication which, in case the WoT is not offering enough confirmations, is performed through the SMP. This is the only real big difference from classical OSNs and it is therefore a feature new for most of the users.

If Bob wants to become friend of Alice he has to go to her public profile page and send her the friendship request. Bob is then asked to input a question, with its relative answer, that will be used by the SMP to authenticate Alice's public key. Figure 6.3 shows the form displayed to Bob. As the form reminds, the answer «should be known only to you (Bob) and Alice», this is fundamental to prevent an attacker from impersonating Alice.

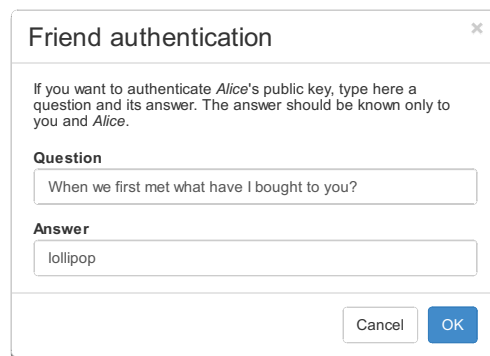
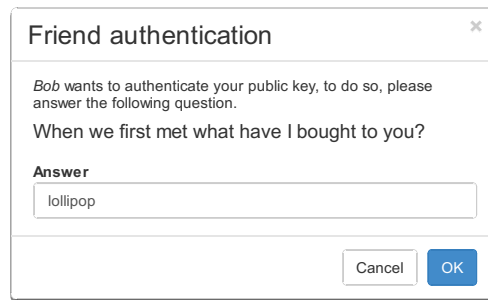


Figure 6.3: SMP in action: authentication of Alice's public key

In the previous figure there is a good example of question and answer since it involves a fact that is most likely known only to Alice and Bob.

When Alice logs into her account she will receive a notification containing Bob's request. Figure 6.4 on page 87 shows the form displayed to Alice, where she has to input the correct answer to prove Bob that she really is Alice.



Friend authentication

Bob wants to authenticate your public key, to do so, please answer the following question.

When we first met what have I bought to you?

Answer

lollipop

Cancel OK

Figure 6.4: SMP in action: answer to Bob's question

If Alice correctly answers to Bob's question the friendship is established otherwise the whole process has to be repeated.

Chapter 7

Experimental Evaluation

In this chapter we present a performance analysis of the various cryptographic primitives offered by the WebCryptoAPI, then we evaluate the overheads introduced in SNAKE due to all the performed cryptographic operations.

7.1 WebCrypto API implementations

All the most famous rendering engines used by current web browser are implementing the WebCrypto API specification and will realistically publish new versions that support that API within a few months. Unfortunately none of implementations offers all the functionalities that our application requires, for the time being we used the two polyfills mentioned in the Chapter 2 to run our application. We ran a series of benchmarks to measure the performance of those two implementation of the WebCrypto API to evaluate what are the differences in terms of execution time for all the cryptographic operations that our application uses. With these tests we want to determine if they are acceptable or not.

To measure the execution time we saved the timestamp right before launching and just after the termination of every operation. Therefore, results are for the whole operation and not only for the computation related to the cryptography part. We chose this approach as the overhead introduced by the asynchronous nature of the API must be taken into consideration.

Unlike many others, we did not acquire the timestamps using the well known JavaScript function `Date.now`. By using the `Date` object the timestamp that we get has a limited resolution, only 1 ms, and a poor accuracy. For

this reason we deemed it inadequate for measuring the execution time of certain operations and chose to use the High Resolution Time API [68]. This API provides a method (`window.performance.now`) that produces a timestamp with a sub-millisecond resolution and higher accuracy which lead to more statistically significant measures.

The tests were executed using Chromium 30.0.1599.101¹ running on a Linux x86_64 3.11² platform equipped with a Intel Core i5-2520M³ processor, the NfWebCrypto plugin was compiled using GCC 4.8.2⁴ and uses OpenSSL 1.0.1e. Each test has been run 100 times to mediate the variation of execution times.

Now we present a performance analysis for each type of cryptographic operation that SNAKE uses.

7.1.1 Key generation and key derivation

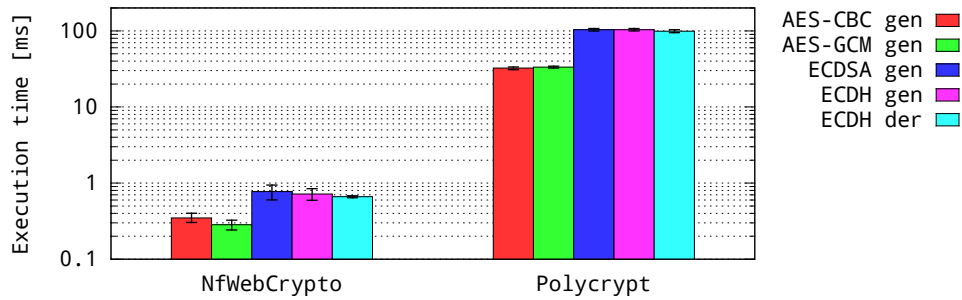


Figure 7.1: Execution times for key generation and key derivation functions

Key generation functions are used to generate cryptographic keys. This methods are widely used in the application: every time a new user, group, profile or friend is created a new key is generated, rekeying operations require the generation of new keys, registration and so on.

Key derivation, using the ECDH algorithm, is an operation that is performed only during friendship establishment.

¹<http://www.chromium.org/>

²<https://www.kernel.org/>

³<http://ark.intel.com/products/52229/>

⁴<http://gcc.gnu.org/>

We executed the test for the generation of AES-CBC and AES-GCM symmetric keys of 256 bits and ECDSA and ECDH keypairs over the curve P-256 and the derivation of a AES-GCM key using ECDH.

Results are shown in Figure 7.1 on page 90. NfWebCrypto is about 100 faster than polycrypt for the key generation and derivation.

This test is a good measure of the overhead for calling a function of the API. The generation of a symmetric key is not a computationally intensive operation, it is only necessary to collect enough entropy (32 bytes in our case) and then return the result to the caller function. We can observe that the overhead for the PolyCrypt implementation is roughly 30 ms. This value is quite high but is justified by the fact that every call to a function of the API creates a dedicated WebWorker that performs the operation. For the NfWebCrypto implementation the overhead is almost negligible since it is well below 1 ms.

The previous considerations cannot be applied to the generation of asymmetric keypairs. In fact, while the generation of a private elliptic curve key is equivalent to the generation of a larger random number, the corresponding public key needs to be computed.

There is a significant difference between the variance of the results, the plugin is quite inconsistent compared with the PolyCrypt. This different behavior is well explained by looking at the implementations of the two polyfills. NfWebCrypto uses OpenSSL functions to get cryptographically strong random bytes using the system entropy pool, while PolyCrypt uses a RC4-based PRNG. Running this tests many times can rapidly deplete the entropy available in the pool, leading to a delay in the generation of the key. Another confirmation in support to this explanation is the key derivation using the ECDH algorithm: the operation performed by this type of derivation is identical to the computation of the public key starting from the private key. In fact the execution time is almost the same but its variance is way lower because the derivation does not need to create a random number.

7.1.2 Encryption and decryption

Symmetric encryption and decryption performances are relevant metrics, since they are the most used cryptographic primitive in our application. We

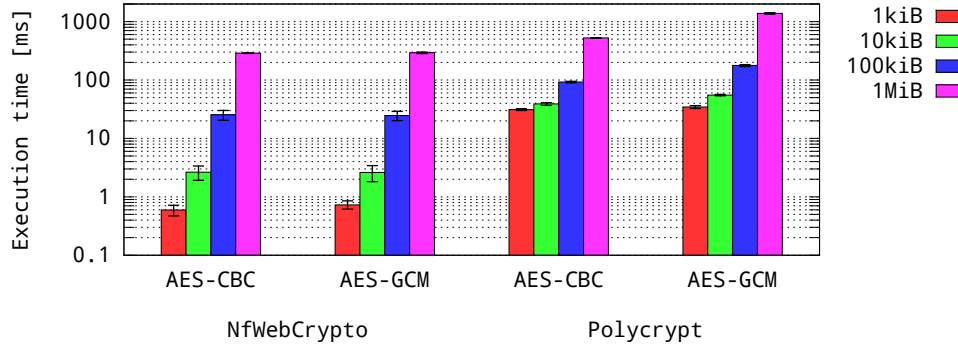


Figure 7.2: Execution times for encryption

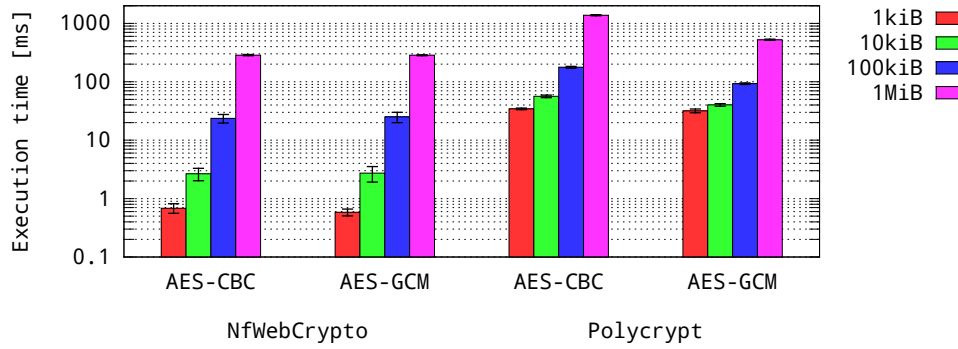


Figure 7.3: Execution times for decryption

executed the test for both algorithms with key lengths of 256 bits and using different sizes of plaintext and ciphertext. The results for encryption are shown in Figure 7.2 and for the decryption in Figure 7.3.

NfWebCrypto has a much higher throughput, compared to PolyCrypt, especially when encrypting or decrypting a small amount of data. The difference is less noticeable when dealing with big plaintexts or ciphertexts and can be explained by looking again at the implementation. PolyCrypt uses directly plaintexts and ciphertexts without making any type of conversion on it. On the other side, since NfWebCrypto communicates with the underlying plugin through JSON messages, all ciphertexts and plaintexts have to be converted from and to Base64 strings. There is therefore an overhead caused by the conversion from typed array to Base64 string and viceversa that limits the throughput of the NfWebCrypto implementation.

7.1.3 Hashing and HMAC

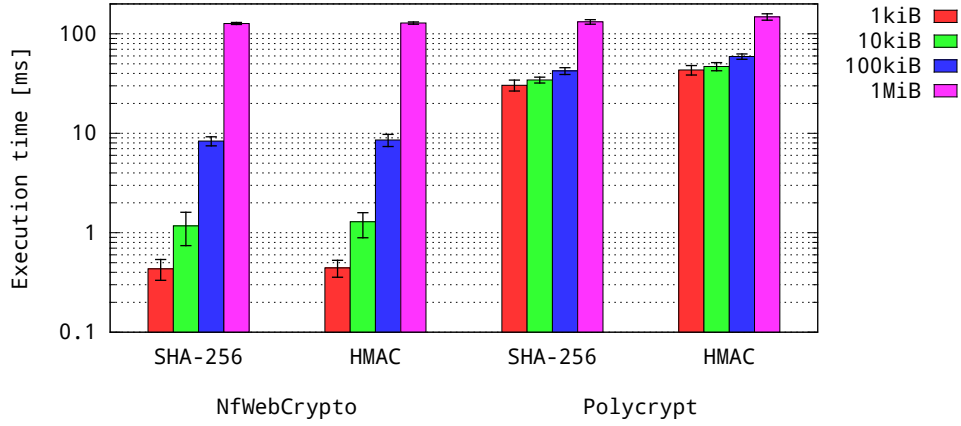


Figure 7.4: Execution times for hashing and HMAC

Hash and MAC primitives are, per se, two types of operations that are not frequently used, but they play an important role during the establishment of friendship.

We executed the test with different message sizes using SHA-256 as hashing algorithm and HMAC-SHA-256 for the MAC. The results are shown in Figure 7.4.

NfWebCrypto is way faster than the PolyCrypt when hashing small amount of data, but the results becomes comparable on larger chunks of data.

7.1.4 Signing and verification

Along with encryption, signing and verification are the most fundamental operation in our application.

We executed the test using different message size to sign and verify using the ECDSA algorithm with curve P-256 and SHA-256 as hashing function. The results are shown in Figure 7.5 on page 94.

The time needed to perform a signature grows linearly with respect to the dimension of the data to sign using the NfWebCrypto implementation, and the execution times are comparable with the previous test for hashing. This shows that the time taken by the hash dominates the total execution time, leading the time to compute or verify a signature to be negligible. With PolyCrypt the situation is exactly the opposite: computation or verification of a signature for

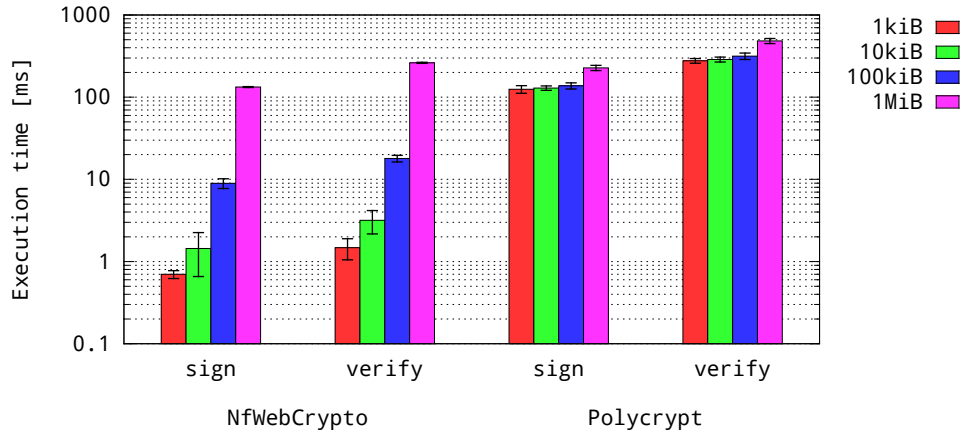


Figure 7.5: Execution times for signing and verification

1 kiB or 10 kiB or 100 kiB of data takes practically the same amount of time, which means the time for hash computation negligible with respect to the signature. As already highlighted in the key generation and derivation tests, performing operations involving asymmetric encryption are very slow using PolyCrypt.

7.1.5 Password based key derivation

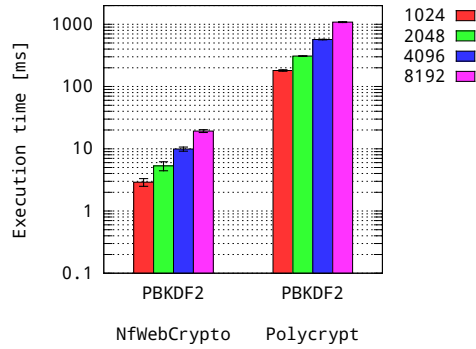


Figure 7.6: Execution times for password based key derivation

Password based key derivation is the operation that derives the key, used to encrypt and decrypt the private descriptor of the user, starting from a password. It is performed once for each registration or login to the application.

We executed the test using SHA-256 and different number of iterations

for the PBKDF2 algorithm. The results are shown in Figure 7.5 on page 94. PolyCrypt is roughly 100 times slower than NfWebCrypto regardless of the number of iterations used.

7.1.6 Final considerations

Table 7.1: NfWebCrypto and Polycrypt benchmark results summary

Operation		NfWebCrypto [ms]				PolyCrypt [ms]			
		1 kiB	10 kiB	100 kiB	1 MiB	1 kiB	10 kiB	100 kiB	1 MiB
AES-CBC	enc	0.595	2.641	25.31	288.4	31.11	38.87	92.25	524.6
	dec	0.684	2.659	23.62	285.4	31.78	40.16	92.81	523.5
AES-GCM	enc	0.729	2.617	24.61	292.1	34.43	55.22	177.1	1380.3
	dec	0.584	2.723	24.98	284.6	34.32	56.25	177.0	1376.1
SHA-256		0.433	1.173	8.347	127.2	30.42	34.37	42.41	132.1
HMAC		0.443	1.291	8.561	128.6	43.30	47.00	59.14	148.3
ECDSA	sign	0.699	1.444	8.953	132.9	124.9	129.2	137.5	227.3
	verify	1.473	3.172	17.94	262.1	277.1	287.9	316.1	483.3
iters		1024	2048	4096	8192	1024	2048	4096	8192
PBKDF2		2.904	5.304	9.896	19.27	181.0	308.6	568.9	1086.0

	Generation	Derivation	Generation	Derivation
AES-CBC	0.348	—	32.49	—
AES-GCM	0.284	—	33.37	—
ECDSA	0.772	—	103.4	—
ECDH	0.719	0.662	103.8	99.04

It is natural to expect that NfWebCrypto is faster than the PolyCrypt, since it relies on an implementation in a compiled language that uses optimized cryptographic primitives. The results is a speed boost of about 10× to 100×, depending on the operation, if compared to a pure JavaScript implementation. However the PolyCrypt is fast enough to be used in our application without noticing high delays.

Both implementations have room for improvement as the source code is not perfectly optimized, but there are some problems that cannot be solved at

all. PolyCrypt suffers of a high overhead per-function call due to the invocation of dedicated WebWorkers and asymmetric cryptography performs pretty slowly, while NfWebCrypto has almost native performance but the throughput is limited by the double conversion required to transfer data from and into the underlying plugin.

In the future there will be support to the WebCrypto API directly in web browser and this will allow to have even greater performances since none of the problems cited before will be present.

With these results we showed that is technologically possible to perform cryptographic computation inside the web browser with adequate performance. All the results showed in the previous figures are summarized in the Table 7.1 on page 95.

7.2 Use cases

With the previous benchmark we saw that is possible to use cryptography in the web browser and the performance are adequate to our application. We now want to quantify the overhead introduced by all the cryptographic functions when performing some actions.

We choose 5 different «use cases» that represent common actions executed by users in a OSN: registration, login, establishment of a friendship, revocation of a friendship and message transmission. These cases cover all the types of basic actions that an user can do within the OSN, all the others are a combination of the previous.

To measure the time spent performing cryptographic operations we used two different configurations: the NfWebCrypto implementation of the WebCrypto API and a dummy implementation. The dummy implementation is an actual implementation of the API but it does not perform any cryptographic operations. For instance, methods for key generation return always a fixed key, encryption functions return the plaintext unmodified and so on. By running each test in both configurations and measuring the time needed to perform the whole operation we can see the amount of time time spent on cryptographic operations.

To simulate a real use case we introduced different network delays between the server and the client. We chose three different network configurations for client-server communication: the loopback interface of the system; a network

where the RTT is 40 ms and a maximum bandwidth of 10 Mbit/s to simulate a domestic connection with a server located in the same continent of the client; a network where the RTT is 100 ms and the maximum bandwidth is 1 Mbit/s to simulate a domestic connection with a server located in a different continent. In the following we will refer to this three configurations using the identifiers NET-LOOPBACK, NET-FAST and NET-SLOW.

We used the same computer to run the server and the client process. The configurations with network delays and bandwidth limitations were realized employing Linux network namespaces⁵, to create different virtual networks, along with Traffic Control⁶, to add the network delay and bandwidth caps.

The tests were executed using a server running Node.js 0.10.21 and MySQL 5.5.33, the configuration of the client is the same of the previous benchmark. An important fact about MySQL was that the database was placed in RAM, placing the folder containing the files used by the database on `tmpfs` file system. With this configuration we setup a server that is fully capable of handling the high load generated by these tests. The assumption that a server for a OSN can perfectly handle the load generated by its users is a very optimistic hypothesis, but we want to show the maximum possible overhead experienced by the user due to all the cryptographic operations, and this occurs only when the server is not overloaded.

As for the previous test, the result are the mean of 100 runs of the same test.

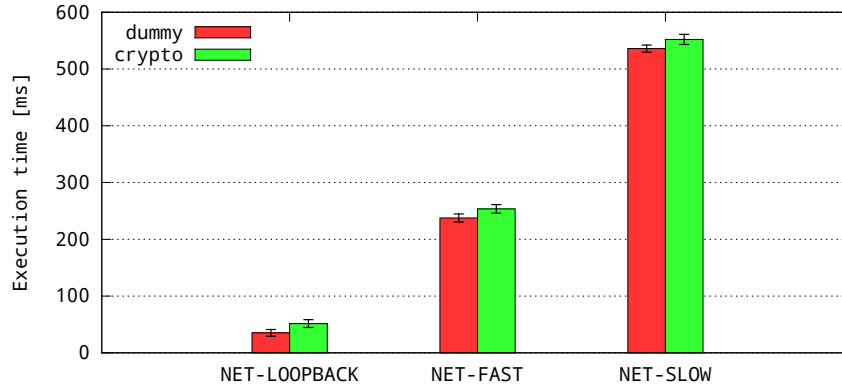
7.2.1 Registration

The first use case is the registration, since it is the first action a user performs to be able to use the application. The registration involves several cryptographic operations: the first one is the derivation of the user master key from the password; then there is the generation of all the symmetric and asymmetric keys; afterwards we create the user descriptor, the public profile and another profile with its associated group of friend.

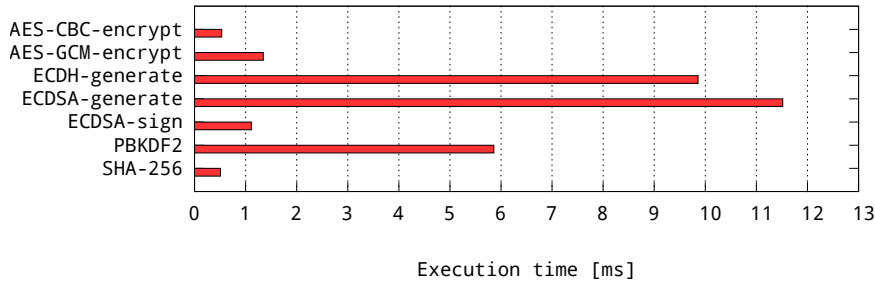
The results of the test are showed in Figure 7.7 on page 98: the difference between using or not the cryptography is roughly 16 ms, regardless of the network configuration.

⁵<http://lwn.net/Articles/219794/>

⁶<http://www.lartc.org/>



(a) Operation time



(b) Cryptographic operations

Figure 7.7: Use case of user registration

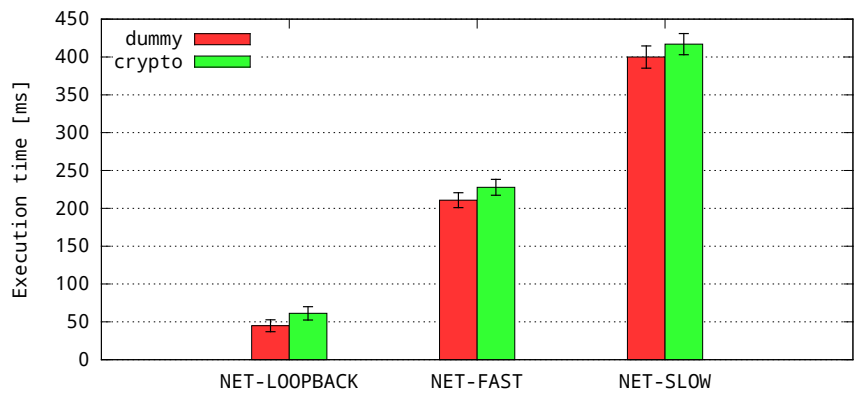
It is not possible to compare the execution times of each cryptographic operation with the results reported in the Subsection 7.1. With the previous benchmark we executed one operation at a time, here we have a lot of operations that are executed simultaneously. Due to the asynchronous nature of JavaScript it is impossible to measure the real time taken by the cryptographic computation as, while an operation is waiting, there may be another one running. Looking at the raw numbers we can see that the time difference between using or not cryptography is 16.35 ms, but if we sum up one-by-one the times of each cryptographic operations we get 30.19 ms. These two different values are not wrong, they are measuring two different times, the former is the difference between using or not cryptography (which is the time we are interested about), the latter is the sum of the time of each operations including delay experienced by every network operation.

The delay on the network increases by a factor of 5 the total execution

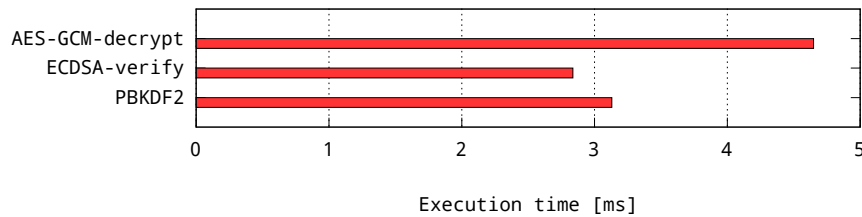
time, this is because during a registration there is an inherent serialization among different phases of the operation. The first request saves the private user descriptor: we could proceed with other operations while we wait for the network operation to complete, but the user ID that the server will give back in the answer of the previous request is needed to create the profiles. The creation of the group of friends associated to the user profile has a similar forced serialization issue.

The overhead experienced by the end user due to cryptographic operations is 31.69%, 6.28%, 2.92%, respectively using the NET-LOOPBACK, NET-FAST and NET-SLOW network configurations.

7.2.2 Login



(a) Operation time



(b) Cryptographic operations

Figure 7.8: Use case of login

This use case measures the time needed to perform a login of a user that has 350 friends, a public profile, another profile only available to friends. The login phase requires to send a request to obtain the ID associated with a username,

then the private descriptor of the user is fetched, and finally the two profiles and the group descriptors are requested.

The results of the test are shown in Figure 7.8 on page 99. In this test too the time required by the cryptographic operations, about 16 ms, is independent from the network configuration.

Depending on the user, the login may require more or less time and to transfer more or less data. The variables that influence this time and the amount of data are there characteristics of the user account, such as his number of friends, of profiles and groups where he is in.

The overhead experienced by the end user due to cryptographic operations is 26.72%, 7.48%, 4.09%, using respectively the NET-LOOPBACK, NET-FAST and NET-SLOW network configurations.

7.2.3 Sending Messages

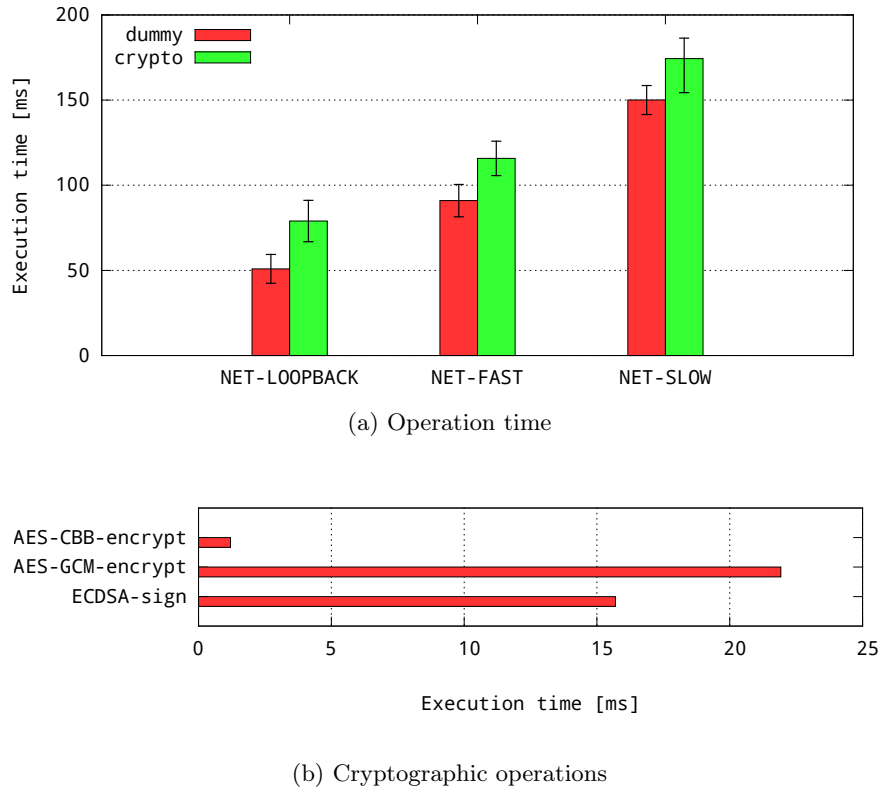


Figure 7.9: Use case of sending messages

Another common operation that is performed in an OSN is sending messages to friends. This test measures the time needed to send 10 message with a payload of 1 kiB to 10 different friends. Sending a message is a simple operation: the message signature is first computed and then the message is encrypted.

The results of the test are showed in Figure 7.9 on page 100, as for the first two tests the time required by the cryptographic operations is independent from the network. This test is an example of an operation that has no serialization points: the 10 messages can be sent all at once independently. Therefore, the difference between the execution in the three network configurations is exactly the network delay⁷.

The overhead experienced by the end user due to cryptographic operations is 33.00%, 21.40%, 13.9%, using respectively the NET-LOOPBACK, NET-FAST and NET-SLOW network configurations. These overheads are greater than the previous ones, especially those for the NET-FAST and NET-SLOW configurations, since without serialization points network time is less incisive and makes the time spent in cryptographic primitive computation more evident.

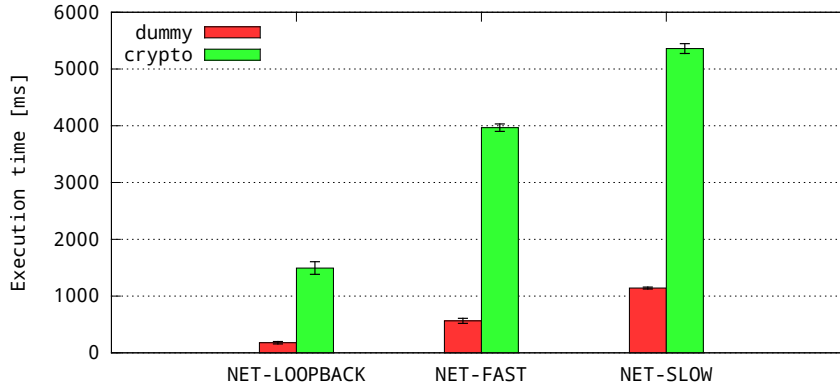
A similar test can be performed for the reception of messages, the result will be almost identical because the operation involved are exactly symmetrical: send a message instead of receiving it, decrypt the message instead of the encrypting it, verify the signature of message instead of computing it.

7.2.4 Establishment of a friendship

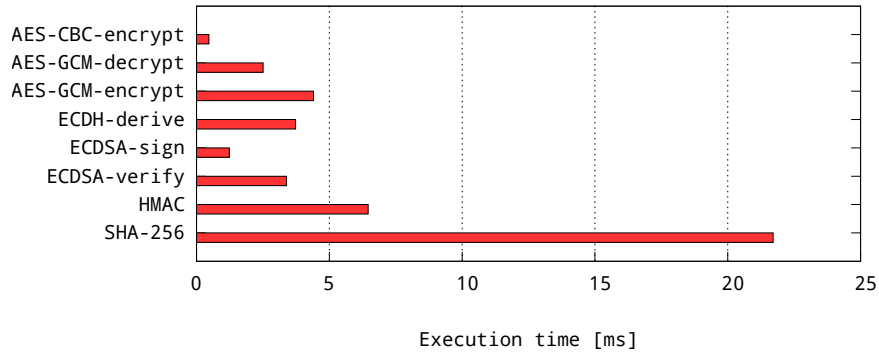
Friendship deserves a dedicated use case because it involves a complicated cryptographic procedure that allows the authentication of public keys of the involved user. The procedure has been already explained in detail in Chapter 3. For this particular test the dummy implementation finalizes the friendship establishment by exchanging a single message, rather than 5.

The friend was added to a group containing 350 users. The results of the test are showed in Figure 7.10 on page 102: in this case the differences between the execution times are not the same but depend on the network configuration, in particular they vary with the network delay between the server and the client.

⁷NET-FAST – NET-LOOPBACK is 115.76 ms – 75.99 ms that is equal to 40 ms and NET-SLOW – NET-LOOPBACK is 174.38 ms – 75.99 ms that is equal to 100 ms.



(a) Operation time



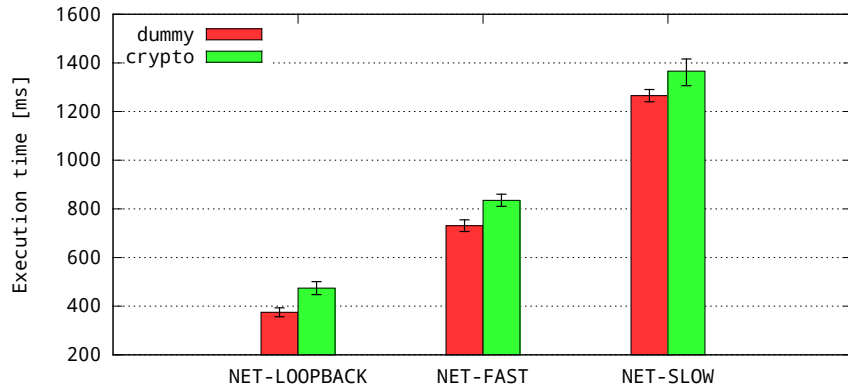
(b) Cryptographic operations

Figure 7.10: Use case of friendship establishment

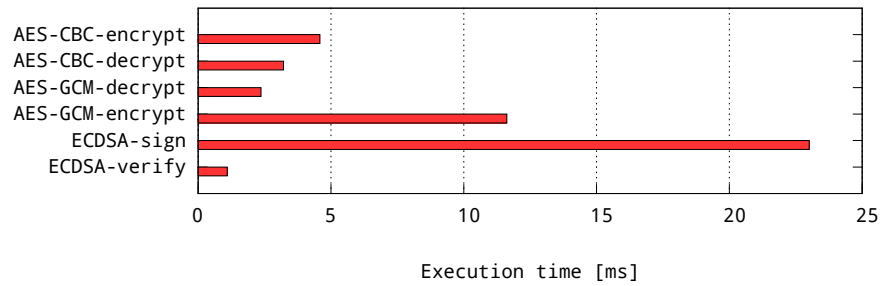
The overhead is significant, the whole operation is 5 to 8 times slower, and this is caused by the greater number of exchanged messages (5 vs 1) introducing serialization points and operations of the FHEMQV-C and SMP which are performed in JavaScript. Considering the properties that our protocol guarantees, we consider this overhead more than acceptable.

7.2.5 Revocation of a friendship

Revocation of a friendship, which mainly consists in the removal of member from a group, is another complex operation that needs to be analyzed. The administrator has to download the administrative data of the group, remove the user from the key graph, and then save it back. Subsequently the list of group members is updated and a rekeying message is sent to the group (that



(a) Operation time



(b) Cryptographic operations

Figure 7.11: Use case of friendship revocation

is also read by the administrator itself) and a message to the old user that notifies his removal.

Removal was performed on a group containing 350 users, the results of the test are showed in Figure 7.11. In this test the number of messages exchanged, either using encryption or not, is the same. The time spent in cryptographic operations is always the same, about 100 ms.

The overhead experienced by the end-user due to cryptographic operations is 20.88%, 12.48%, 7.40%, using respectively the NET-LOOPBACK, NET-FAST and NET-SLOW network configurations.

7.2.6 Final considerations

All the results showed in the previous figures are summarized in Table 7.2 on page 104.

With these tests we showed that by encrypting and signing every entity,

Table 7.2: Use cases benchmark results summary

Use case	NET-LOOPBACK [MS]		NET-FAST [MS]		NET-SLOW [MS]	
	crypto	dummy	crypto	dummy	crypto	dummy
Registration	51.565	35.219	253.52	237.57	552.06	535.94
Login	61.283	44.905	227.81	210.752	416.95	399.88
Friendship	1493.8	177.38	3966.3	563.42	5360.6	1141.1
Remove friend	454.07	375.07	835.15	730.93	1366.3	1265.2
Send messages	75.997	50.913	115.76	90.989	174.381	150.05

using a protocol for securing friendship establishment, using another protocol to manage groups of user some overheads are introduced.

It is important to remember that the server was offering the highest possible performance, therefore the overheads here reported are an upper bound.

Excluding the NET-LOOPBACK configuration the overhead experienced using the application in a real network is between 3% and 14% for common operations such as registration, login, send or receive messages. The removal of a member from a group is an operation seldom performed which however does not introduce an excessive overhead. The only exception is the establishment of a friendship that is from 5 to 8 times slower using our protocol compared to a naïve solution, anyway the time required to perform that operation is more than adequate.

Chapter 8

Future developments

In this chapter we analyze several aspects of SNAKE that need attention or provide a starting point for future developments.

8.1 System architecture

To improve the usability of the system, we want to extend the usage of the Web of Trust reintroducing transitivity of trust. This means we have to consider, not only public key authentications made by directly authenticated friends, but also authentications by friends of friends and so on. Since we do not see a viable way to completely remove the information leakage about trust evaluations described in Subsection 3.4.3.2, we will direct our efforts towards finding an appropriate trade-off between leakage of trust evaluations and usefulness of the WoT.

Once we have a stable system, we plan to proceed with one of the primary objectives of SNAKE: building a platform for secure and private one-to-one and many-to-many message exchange that can be easily reused by third party applications. These applications can be untrusted and completely unaware of the underlying cryptographic system: they will interact with the main module of SNAKE through a restricted API, thanks to the Messaging API [34], from a sandboxed environment. The user will decide what permissions each application has and what profile provide to it.

The most eminent examples of such applications are an online collaborative office suites and a file (or photo) sharing application. Thanks to technologies such as WebODF [40] we will be able to edit ODF documents in a collabora-

tive and, for the first time, secure and private way with zero configuration and without major performance drops. For what concerns the file sharing application, we plan to implement it using WebRTC [55] for P2P transfer or relying on third party services such as Mega.co.nz [63]. In both case the exchanged data will be symmetrically encrypted with a key exchanged over a secure channel established through SNAKE.

8.2 Storage server

For what concerns the storage infrastructure, we aim to improve its scalability allowing multiple distinct and independently managed storage providers to interoperate. The idea is to introduce a suffix to record identifiers to create a server-level namespace. For instance, the client will not query the storage server for message with identifier 7011, but for the message with identifier 7011@raven.com, where raven.com is the address of another storage server. This *federation* approach does not only improve scalability, but also reduces the view of a single storage server over the data, mitigating the risks of the MALICIOUS-SERVER scenario. On the other hand we also need to provide anti-DoS measures for the storage servers to protect them from abuses by malicious users.

Another aspect that needs attention is the fact that the creator of a record on the remote storage has full control over it, and can delete or update it without being noticed by other users. He can also alter attributes such as the creation date, since the storage server does not keep timestamps, for the reasons we saw in Section 5.2. Two viable solutions are worth being explored. The first consists in removing the possibility to delete a record, let the storage server keep trace of the history of each record and make it available to users. The other option is to introduce some kind of sorting strategy managed on the client-side. Both approaches introduce several new challenges that need to be analyzed.

8.3 Messages

For what concerns the messaging system, we plan to introduce a real-time chat through the usage of push notifications, which can be easily integrated in

Socket.IO [86]. However, this introduces new issues from the scalability point of view and in presence of a distributed storage infrastructure.

Message format is another part of the design of SNAKE which needs improvement. In fact, currently, the choice of the employed cryptographic primitives are implicit over all the system, while in future it might be useful to be able to upgrade them and specify which ones are being used on a per-message level. While this would make our system more future proof, it also considerably increase the attack surface as it may provide the possibility of performing ciphersuite downgrade attacks. Moreover, if multiple algorithms are allowed, the various clients have to support all of them and must enforce a policy on allowed and forbidden algorithms, which is not always easy to define and is subject to security trade-offs.

Another feature that would fit well in a chat application is having an *OTR-like mode*, where the properties illustrated in Subsection 1.1.2 are guaranteed. This can be achieved quite easily, it is enough to disable message signing and negotiating a new symmetric key at each message exchange through a D-H key agreement run.

8.4 Groups

We proposed a solution to efficiently manage the creation and the evolution of groups in our system, the main focus was to create and implement the fundamental aspects for group management. However, there are some aspects that can be improved introducing simple but effective changes.

One of this aspects is group administration. It would be interesting to investigate a more democratic group management. In fact, at current state, each administrator can take over the group and autonomously alter each attribute of its descriptor. Through cryptographic threshold schemes it would be possible to require at least t administrators agree on a change to make it effective.

We also plan to introduce several optimizations for the rekeying process. For instance, every action that involves modifying the participants list, currently requires an administrator to handle them. It is possible that the administrator has to process multiple requests at the same time. At current status, a list of requests is processed sequentially one after the other. Handling multiple requests at the same time can reduce the the rekeying costs. However,

we do not want to introduce what in literature is called *batch rekeying* (introduced by [61] and revised in other subsequent works), since this approach uses a periodic rekeying that allows a removed member to continue to read the conversations of the group for a period of time after his removal.

The applicable improvements are mainly two. The first one is to handle at same time the insertion of multiple users, for instance if there are 4 join requests they can be processed together modifying only once the `adminData` field of the group descriptor. The second one consists in avoiding to process useless requests: if a user asked to get access to a group and then he also requests to leave the group (maybe due to a mistake), it is useless to add it and then remove it. This possibly «phantom» requests can be ignored and therefore 2 rekeying messages can be saved.

Another possible extension of the current group management system is introducing the concept of *subgroup*. Sometimes it is necessary to send a message only to a subset of the participants of a group, with the solution that we proposed it is necessary to create a new group, but, with a simple modification to the tree of users, it is possible to efficiently communicate with only part of the group. If communication with a specific subgroup is frequent, this technique can be used to invite the interested participants to a new fully featured group. This is particularly useful to efficiently create a new group identical to the previous one except for a couple of members.

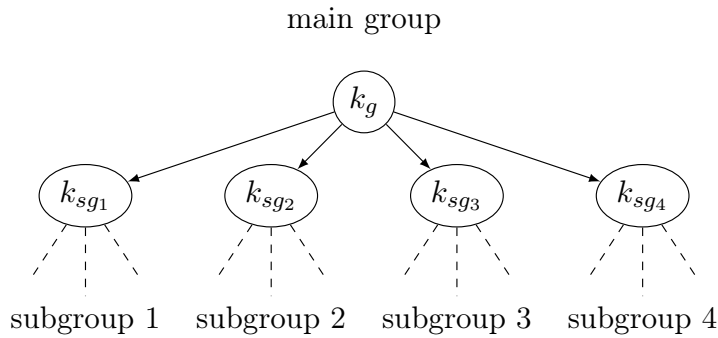


Figure 8.1: Subgroups

Another approach to communicate with subgroups consists in splitting a bigger group. It is possible to reserve the first level of the tree to define a set of subgroups as showed in Figure 8.1. Each subgroup is handled as a normal group. Since our tree has degree 7, a group can only have 7 subgroups, if

more are needed, it is sufficient to reserve 2 levels of the tree to have up to 49 subgroups.

We can imagine our group of friend composed with different subgroups, i.e. best friends, school mates and so on. Communication with the group is encrypted using the key k_g , while communication with subgroups is achieved by using the proper subgroup key (i.e. k_{sg1} or k_{sg2}).

Conclusions

In this work we proposed, and implemented, a privacy-aware end-to-end encrypted OSN. We mainly focused on four aspects that we consider fundamental: an in-band method for the authentication of user's public keys, an efficient method to manage secure groups and a storage strategy that provides the anonymity of user's relationships when data is at rest. All this in a user-friendly web application similar to current OSNs from the end user's point of view.

First, we explored the state of the art of OSNs. We studied the most common end-to-end encryption protocols that are currently available, PGP and OTR. We also considered in detail the various types of system architecture presented in the literature and in real systems, paying particular attention to some crucial aspects such as: public key authentication, communication and management of groups of users and anonymity of user's data.

We described the architecture of our system, detailing the main components and how they interact. We have defined a protocol for friendship establishment that allows in-band authentication of public keys. The solution employs the FHEMQV-C protocol as key agreement algorithm along with SMP and a Web of Trust for public key authentication. For the management of groups of users we used a key graph approach, this approach allowed us to obtain a rekeying cost that is logarithmic in the group size. We designed the whole system to preserve anonymity of the social graph, in the case of an honest storage server.

Finally, we performed an evaluation of the overhead that the user will experience due to the various cryptographic operations showing that they are fully acceptable and do not compromise the user experience.

In conclusion, our work has shown that is possible to have all the feature of current OSNs with an end-to-end encrypted solution that is able to preserve the anonymity of user's data.

Conclusions

Acronyms

ACL Access Control List, *Glossary*: ACL.

AES Advanced Encryption Standard, *Glossary*: AES.

API Application Programming Interface, *Glossary*: API.

CA Certification Authority, *Glossary*: Certification Authority.

CBC Cipher-block Chaining, *Glossary*: CBC.

CRUD Create, Read, Update and Delete, *Glossary*: CRUD.

D-H Diffie-Hellman key exchange.

DHT Distributed Hash Table, *Glossary*: DHT.

DoS Denial of Service, *Glossary*: Denial of Service attack.

ECC Elliptic Curve Cryptography, *Glossary*: Elliptic Curve Cryptography.

ECDH Elliptic Curve Diffie-Hellman, *Glossary*: ECDH.

ECDSA Elliptic Curve Digital Signature Algorithm, *Glossary*: ECDSA.

FHMQV Fully Hashed Menezes-Qu-Vanstone, *Glossary*: Fully Hashed Menezes-Qu-Vanstone.

FOSS Free Open Source Software.

GCM Galois/Counter Mode, *Glossary*: GCM.

HMAC keyed-Hash Message Authentication Code, *Glossary*: HMAC.

HTTP Hypertext Transfer Protocol, *Glossary*: HTTP.

HTTPS Hypertext Transfer Protocol Secure, *Glossary*: HTTPS.

IV Initialization Vector.

JSON JavaScript Object Notation, *Glossary*: JSON.

MAC message authentication code, *Glossary*: message authentication code.

MITM man-in-the-middle, *Glossary*: MITM attack.

NIST National Institute of Standards and Technology, *Glossary*: NIST.

ODF Open Document Format for Office Applications, *Glossary*: Open Document Format for Office Applications.

OSN Online Social Network, *Glossary*: Online Social Network.

OTR Off-the-Record Messaging, *Glossary*: Off-the-Record Messaging.

P2P Peer-to-peer, *Glossary*: P2P.

PBKDF2 Password-Based Key Derivation Function 2, *Glossary*: PBKDF2.

PGP Pretty Good Privacy, *Glossary*: Pretty Good Privacy.

PKC Public Key Cryptography, *Glossary*: Public Key Cryptography.

PKI Public Key Infrastructure, *Glossary*: Public Key Infrastructure.

PRNG Pseudorandom Number Generator, *Glossary*: PRNG.

RDBMS Relational Database Management System, *Glossary*: RDBMS.

RSA Rivest Shamir Adleman, *Glossary*: RSA.

SHA Secure Hash Algorithm, *Glossary*: SHA.

SMP Socialist Millionaire Protocol, *Glossary*: Socialist Millionaire Protocol.

TLS Transport Layer Security, *Glossary*: TLS.

URL Uniform Resource Locator, *Glossary*: URL.

WoT Web of Trust, *Glossary*: Web of Trust.

Glossary

ACL a list of permissions that specifies which users or a generic entity are granted access to objects, as well as what operations are allowed on given objects. 11, 14–16, 113

AES a specification by NIST of a symmetric-key encrypting algorithm. It is based on the Rijndael cipher, it has a block size of 128 bits and supports keys with three different key lengths: 128, 192 and 256 bits. 2, 31, 32, 34, 35, 43, 71, 91, 113

API a set of routines which allow to interact with a specific part of a software system. For instance the Berkeley sockets API offer a standard interface to interact with operating system's TCP/IP stack.. 15, 31, 89–91, 96, 105, 113

asymmetric cryptography see Public Key Cryptography. 3, 43, 96

CBC a mode of operation for symmetric key cryptographic block ciphers in which each block of plaintext is XORed with the previous ciphertext block before being encrypted. 32, 43, 71, 91, 113

Certification Authority a part of a PKI which guarantees the authenticity of a public key by digitally signing it. It is a critical and central component which requires full trust by all the users of the PKI. 24, 113

CRUD operations that can be performed with different HTTP requests. Create: POST, Read: GET, Update: PUT, Delete: DELETE. 3, 113

Denial of Service attack attack which aims to make a service unavailable to its users. This kind of attack usually consists in overloading a server with bogus requests. 41, 43, 106, 113

DHT a decentralized distributed system that provides a lookup service similar to a hash table. 15, 113

Diffie-Hellman key exchange an asymmetric cryptographic protocol for key exchange that allows two parties to establish a shared secret key over an insecure communications channel. 8, 9, 31, 38, 41–43, 45, 107

ECDH an anonymous key agreement protocol that uses ECC to establish a shared secret over an insecure channel. 31, 90, 91, 113

ECDSA a Digital Signature Algorithm that uses ECC. 31, 32, 34, 91, 93, 113

Elliptic Curve Cryptography form of asymmetric cryptography based on the algebraic structure of elliptic curves over finite fields. 3, 32, 113

Fully Hashed Menezes-Qu-Vanstone a state-of-the-art cryptographic protocol for authenticated key agreement. 3, 33, 38, 39, 41–43, 45, 46, 48, 102, 113

GCM a mode of operation for symmetric key cryptographic block ciphers designed to provide both data authenticity (integrity) and confidentiality. 32, 34, 35, 71, 91, 113

hashtag a single word or a short phrase without spaces beginning with the # character. It got famous on Twitter [52] to identify the argument of a particular post. 13

HMAC a particular type of MAC that involves the use of a cryptographic hash function, such as SHA, in combination with a secret cryptographic key. 31, 32, 93, 113

HTTP an application protocol used to serve the contents of the World Wide Web.. 19, 24, 27, 28, 114

HTTPS the secure version of HTTP. It is the result of encapsulating an HTTP communication in a TLS stream. 19, 23, 25, 114

in-band in authentication, in-band refers to utilizing only one communication channel both to exchange cryptographic keys and to communicate. 37

Initialization Vector a random fixed-size input used by some block cipher modes of operation to initialize the encryption or decryption. It is used to obtain different ciphertexts when encrypting more than once the same plaintext. 34, 71

JSON an open standard format used to transmit data objects consisting of attribute-value pairs expressed in a human readable format. 28, 33–35, 70, 92, 114

message authentication code a short piece of information used to guarantee integrity and authenticity of a message. It is usually computed through a keyed cryptographic hash. 9, 32, 39, 45, 51, 93, 114

MITM attack an attack where a malicious user intercepts, decrypts and re-encrypts the data passing over a secure communication channel pretending to be one end of the communication to the other end. This kind of attack is usually completely transparent to legitimate users, since the traffic is forwarded to the legitimate recipient and the system apparently works properly. This kind of attack usually relies on fake public keys, therefore proper public key authentication is required to prevent it. 6, 114

NIST a U.S. federal technology agency that works with industry to develop and apply technology, measurements, and standards. In particular it develops, maintains and promotes a number of standards and guidance that cover a wide range of cryptographic technology standards. 32, 33, 42, 43, 114

Off-the-Record Messaging a protocol to end-to-end encrypt conversations taking place over another IM protocol such as Skype or the Facebook Chat. Its specifications are defined in [2, 13]. 2, 4, 5, 7, 9, 16, 18, 42, 107, 114

Online Social Network an online platform which aims to build social relations between people. Their main features usually consist in one-to-one communication, content sharing and so on. 1, 2, 5, 9–15, 17–21, 59, 69, 83, 84, 86, 96, 97, 101, 114

- Open Document Format for Office Applications** a ISO/IEC standard file format for spreadsheets, charts, presentations and word processing documents. 105, 114
- out-of-band** in authentication, out-of-band refers to the practice of using two separate communication channels, one secure channel is used to exchange some cryptographic keys that are used to ensure a secure communication over the other channel. 2, 9, 15, 16, 48
- P2P** a type of decentralized and distributed network architecture where each node, which is called *peer*, behaves both as consumer and supplier of contents. This model is an alternative to the classical client-server approach. 14–17, 19, 106, 114
- PBKDF2** a key derivation function that applies a pseudorandom function (cryptographic hash, HMAC, ...) to an input password along with a salt value and repeats the process many times to produce a cryptographic key. 31, 32, 95, 114
- polyfill** a library that provides a, usually standard, technology that has not been implemented in version of a web browser. These libraries are used to add new features to old web browsers or to provide experimental features to current ones. 26, 32, 89, 91
- Pretty Good Privacy** a famous program for data encryption and signing, mostly used for e-mail encryption and signing. It first introduced the concept of WoT. Its format has been standardized in [17]. 2, 5–7, 11, 15, 18, 49, 114
- PRNG** an algorithm for generating a sequence of numbers that approximates the properties of random numbers. 31, 65, 91, 114
- proxy re-encryption** cryptographic scheme where a third-party, the proxy, transforms a ciphertext encrypted for one party into another ciphertext, so that it may be decrypted by another party. 10, 13, 14, 17
- Public Key Cryptography** a family of cryptographic algorithms which uses a keypair, i.e. a private (secret) key and a public key. Usually the public key is used for encryption and signature verification, while the private part is used for decryption and signing. 6, 7, 14, 37, 114, 117

Public Key Infrastructure an infrastructure to manage creation, distribution and revocation of digital certificates in system using PKC. It is formed by various entities, including CA. 5, 15, 24, 114

RDBMS a database management system that is based on the relational model. 24, 114

RSA a standard asymmetric cryptographic algorithm used to sign, verify, encrypt and decrypt data. It was invented by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977. 8, 13, 31, 33, 114

SHA a set of cryptographic hash functions published by the NIST. 31, 32, 39, 93, 94, 114, 118

Socialist Millionaire Protocol a protocol used to verify whether the two ends of the posses the same secret value, without revealing it.. 2, 9, 33, 37, 42, 45–48, 52–56, 86, 87, 102, 114

TLS a cryptographic protocol used to achieve secure communications over the Internet. It employs the use of asymmetric cryptography for the authentication and the exchange of symmetric-key used for the communication. 19, 23–25, 114

URL a text string identifying a resource. A typical example is a web address, for example `http://server/path/image.jpg`. 27, 115

URL fragment identifier part of a URL after the `#` symbol. Its main use consists in identifying a particular part of a document, for instance *friends* in `http://osn.com/mark/profile.html#friends` might indicate the part of the web page where the list of friends of a user is. When using HTTP, the fragment identifier is never sent to the server, it is handled by the client. 27

Web of Trust a concept used to indicate a graph whose nodes are users and edges represent the fact that a user trusts another user. In the context of public key authentication, an edge means that a user verified that a public key actually belongs to the declared owner. 1, 2, 6, 7, 25, 37, 42, 48, 49, 51–56, 86, 105, 115

whistleblower the person who exposes a misconduct or an illegal activity occurring in a private or public organization. 2

zero-knowledge proof a proof where one party, known as the *prover*, can prove to another party, known as the *verifier*, to possess a certain piece of information, without revealing it. 16

Bibliography

- [1] Spencer Ackerman. Lavabit email service abruptly shut down citing government interference. <http://www.theguardian.com/technology/2013/aug/08/lavabit-email-shut-down-edward-snowden>, aug 2013.
- [2] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, WPES '07, pages 41–47, New York, NY, USA, 2007. ACM.
- [3] Jeremy Ashkenas. Backbone.js. <http://backbonejs.org/>.
- [4] Jeremy Ashkenas. Underscore.js. <http://underscorejs.org/>.
- [5] Michael Backes, Matteo Maffei, and Kim Pecina. A Security API for Distributed Social Networks. In *Proc. Network and Distributed System Security Symposium (NDSS'11)*, pages 35–51. Internet Society, 2011.
- [6] Randolph Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: an online social network with user-defined privacy. In *SIGCOMM*, pages 135–146, 2009.
- [7] Richard Barnes. FoxyCrypt. <https://github.com/polycrypt/foxcrypt>.
- [8] Richard Barnes. PolyCrypt. <http://polycrypt.net/>.
- [9] Filipe Beato, Markulf Kohlweiss, and Karel Wouters. Scramble! Your Social Network Data. In *PETS*, pages 211–225, 2011.
- [10] Mihir Bellare and Adriana Palacio. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. pages 273–289. Springer-Verlag, 2004.

- [11] Robin Berjon, Aryeh Gregor, Anne van Kesteren, Ms2ger, and Alex Russell. W3C DOM4 (Promises). Technical report, nov 2013. <http://www.w3.org/TR/2013/WD-dom-20131107/#promises>.
- [12] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, jul 1970.
- [13] Nikita Borisov. Off-the-record communication, or, why not to use PGP. In *In WPES '04: the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84. ACM Press, 2004.
- [14] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A Fair and Efficient Solution to the Socialist Millionaires' Problem. *Discrete Applied Mathematics*, 111:2001, 2001.
- [15] Joe Bowser, Michael Brooks, Rob Ellis, Dave Johnson, Anis Kadri, Brian Leroux, Jesse MacFadyen, Filip Maj, Eric Oesterle, Brock Whitten, Herman Wong, Shazron Abdullah, and Adobe Systems. PhoneGap. <http://phonegap.com/>.
- [16] Sonja Buchegger, Doris SchiÅsberg, Le-Hung Vu, and Anwitaman Datta. PeerSoN: P2P social networking: early experiences and insights. In *SNS*, pages 46–52, 2009.
- [17] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and F. Thayer. RFC 4880 - OpenPGP Message Format. Technical report, nov 2007.
- [18] Ran Canetti, Juan Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. pages 708–716, 1999.
- [19] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. pages 453–474. Springer-Verlag, 2001.
- [20] Ran Canetti and Hugo Krawczyk. Universally Composable Notions of Key Exchange and Secure Channels, 2002. <http://eprint.iacr.org/>.
- [21] Microsoft Corporation. Microsoft CryptoAPI. <http://msdn.microsoft.com/en-us/library/Windows/desktop/aa380255.aspx>.

- [22] Microsoft Corporation. Web Cryptography (IE11). <http://msdn.microsoft.com/en-us/library/ie/dn302338.aspx>.
- [23] Oracle Corporation. MySQL. <https://www.mysql.com/>.
- [24] David Dahl and Ryan Sleevi. Web Cryptography API. Technical report, jun 2013. <http://www.w3.org/TR/2013/WD-WebCryptoAPI-20130625/>.
- [25] Ryan Lienhart Dahl, Node.js Developers, and Joyent. Node.js. <http://nodejs.org/>.
- [26] John-David Dalton, Blaine Bublitz, Kit Cambridge, and Mathias Byens. Lo-Dash. <http://lodash.com/>.
- [27] S. De Capitani di Vimercati, S. Foresti, S. Paraboschi, G. Pelosi, and P. Samarati. Efficient and Private Access to Outsourced Data. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 710–719, 2011.
- [28] E. De Cristofaro, C. Soriente, G. Tsudik, and A. Williams. Hummingbird: Privacy at the Time of Twitter. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 285–299, 2012.
- [29] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, sep 2006.
- [30] Dmytro Dogadailo. Recursive JSON (RJSON). <https://github.com/dogada/RJSON>.
- [31] Longzhi Du, Rui Kong, Fengxian Ren, Jianbin Hu, and Zhong Chen. PrivOSN: Practical Privacy in Online Social Network. In *2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*, page 466, 2011.
- [32] Ariel J. Feldman. Presentation of "Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider". <https://www.youtube.com/watch?v=mNdXCKAQ4KM#t=1539>, 2012.
- [33] Ariel J. Feldman, Aaron Blankstein, Michael J. Freedman, and Edward W. Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. 2012.

BIBLIOGRAPHY

- [34] Jose Manuel Cantera Fonseca, Eduardo Fulla, and Zoltan Kis. Messaging API. Technical report, may 2013. <http://www.w3.org/TR/2013/WD-messaging-20130516/>.
- [35] The Diaspora Foundation. The Diaspora Project. <https://diasporafoundation.org/>.
- [36] Gary Court Francis Galiegue, Kris Zyp. JSON Schema: core definitions and terminology. <http://tools.ietf.org/html/draft-zyp-json-schema-04>, jan 2013.
- [37] Gary Court Francis Galiegue, Kris Zyp. JSON Schema: interactive and non interactive validation. <http://tools.ietf.org/html/draft-fge-json-schema-validation-00>, feb 2013.
- [38] Sadayuki Furuhashi. MessagePack. <http://msgpack.org/>.
- [39] Marcio Galli, Roger Soares, and Ian Oeschger. Inner-browsing extending the browser navigation paradigm. https://developer.mozilla.org/en-US/docs/Inner-browsing_extending_the_browser_navigation_paradigm, May 2003.
- [40] KO GmbH. WebODF. <http://webodf.org/>.
- [41] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 182–194, New York, NY, USA, 1987. ACM.
- [42] Justin Goshi and Richard E. Ladner. Algorithms for Dynamic Multicast Key Distribution Trees. In *in Proc. 22nd Symposium on Principles of Distributed Computing (PODC)*, pages 243–251. ACM, 2003.
- [43] Kalman Graffi, Patrick Mukherjee, Burkhard Menges, Daniel Hartung, Aleksandra Kovacevic, and Ralf Steinmetz. Practical security in p2p-based social networks. In *LCN*, pages 269–272, 2009.
- [44] Saikat Guha. NOYB: Privacy in Online Social Networks. 2008.
- [45] Joachim Henke. basE91 - binary to ASCII text encoding. <http://base91.sourceforge.net/>.

- [46] Ian Hickson. The WebSocket API. Technical report, sep 2012. <http://www.w3.org/TR/2012/CR-websockets-20120920/>.
- [47] Facebook Inc. Facebook. <https://www.facebook.com/>.
- [48] Google Inc. Google Native Client. <https://developers.google.com/native-client/>.
- [49] Google Inc. Google Play. <https://play.google.com/store>.
- [50] MongoDB Inc. BSON - Binary JSON. <http://bsonspec.org/>.
- [51] Netflix Inc. NfWebCrypto. <https://github.com/Netflix/NfWebCrypto>.
- [52] Twitter Inc. Twitter. <https://twitter.com/>.
- [53] Sonia Jahid, Prateek Mittal, and Nikita Borisov. EASiER: encryption-based access control in social networks with efficient revocation. In *ASI-ACCS*, pages 411–415, 2011.
- [54] Sonia Jahid, Shirin Nilizadeh, Prateek Mittal, Nikita Borisov, and Apu Kapadia. DECENT: A decentralized architecture for enforcing privacy in online social networks. In *PerCom Workshops*, pages 326–332, 2012.
- [55] Cullen Jennings, Anant Narayanan, Adam Bergkvist, and Daniel Burnett. WebRTC 1.0: Real-time Communication Between Browsers. Technical report, sep 2013. <http://www.w3.org/TR/2013/WD-webrtc-20130910/>.
- [56] . Burton S. Kaliski, Jr. An unknown key-share attack on the MQV key agreement protocol. *ACM Trans. Inf. Syst. Secur.*, 4(3):275–288, aug 2001.
- [57] Kenton Powell Ewen MacAskill Ruth Spencer Lisa van Gelder Kenan Davis, Nadja Popovich. NSA files decoded: Edward Snowden’s surveillance revelations explained. <http://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded>.

- [58] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS), 2013.
- [59] Hugo Krawczyk. HMQV: a high-performance secure diffie-hellman protocol. In *Proceedings of the 25th annual international conference on Advances in Cryptology*, CRYPTO'05, pages 546–566, Berlin, Heidelberg, 2005. Springer-Verlag.
- [60] Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas, and Scott Vanstone. An Efficient Protocol for Authenticated Key Agreement. Technical report, 1998.
- [61] Xiaozhou Steve Li, Yang Richard Yang, Mohamed G. Gouda, and Simon S. Lam. Batch Rekeying for Secure Group Communications. In *Proceedings of the 10th International Conference on World Wide Web*, WWW '01, pages 525–534, New York, NY, USA, 2001. ACM.
- [62] Dongtao Liu, Amre Shakimov, Ramón Cáceres, Alexander Varshavsky, and Landon P. Cox. Confidant: protecting OSN data without locking it up. In *Proceedings of the 12th International Middleware Conference*, Middleware '11, pages 60–79, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.
- [63] Mega Ltd. Mega. <https://mega.co.nz/>.
- [64] Haibin Lu. A Novel High-Order Tree for Secure Multicast Key Management. *IEEE Trans. Comput.*, 54(2):214–224, feb 2005.
- [65] Matt Lucas. flyByNight. 2008.
- [66] Geraint Luff. tv4: tiny Validator for JSON Schema v4. <https://github.com/geraintluff/tv4>.
- [67] Wanying Luo, Qi Xie, and Urs Hengartner. FaceCloak: An Architecture for User Privacy on Social Networking Sites. In *CSE (3)*, pages 26–33, 2009.
- [68] Jatinder Mann. High Resolution Time. Technical report, dec 2012. <http://www.w3.org/TR/2012/REC-hr-time-20121217/>.

- [69] MarketingCharts. 18-24-Year-Olds on Facebook Boast an Average of 510 Friends. <http://www.marketingcharts.com/wp/direct/18-24-year-olds-on-facebook-boast-an-average-of-510-friends-28353/>, apr 2013.
- [70] Moxie Marlinspike. Convergence.io. <http://convergence.io/>.
- [71] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11:431–441, 1963.
- [72] A. Menezes, M. Qu, and S. Vanstone. Some new key agreement protocols providing mutual implicit authentication. In *Selected Areas in Cryptography*, 1995.
- [73] Alfred Menezes. Another Look at HMQV. *IACR Eprint archive*, 2005:2005, 2005.
- [74] Alfred Menezes and Berkant Ustaoglu. On the importance of public-key validation in the MQV and HMQV key agreement protocols. In *Proceedings of the 7th international conference on Cryptology in India, INDOCRYPT'06*, pages 133–147, Berlin, Heidelberg, 2006. Springer-Verlag.
- [75] Mozilla, AOL, Red Hat, Sun Microsystems, Oracle Corporation, and Google. Network Security Services. <https://developer.mozilla.org/en/docs/NSS>.
- [76] Rammohan Narendula, Thanasis G. Papaioannou, and Karl Aberer. Privacy-Aware and Highly-Available OSN Profiles. In *Proceedings of the 2010 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, WETICE '10*, pages 211–216, Washington, DC, USA, 2010. IEEE Computer Society.
- [77] Shirin Nilizadeh, Sonia Jahid, Prateek Mittal, Nikita Borisov, and Apu Kapadia. Cachet: a decentralized architecture for privacy preserving social networking with caching. In *CoNEXT*, pages 337–348, 2012.
- [78] Mark Otto and Jacob Thornton. Bootstrap. <http://getbootstrap.com/>.

BIBLIOGRAPHY

- [79] Leonidas Oy. Transparency. <https://github.com/leonidas/transparency/>.
- [80] Mike Perry. Why the Web of Trust Sucks. <https://lists.torproject.org/pipermail/tor-talk/2013-September/030235.html>, September 2013.
- [81] The Chromium Project. Implementation status of Web Crypto API in Chromium. <http://www.chromestatus.com/features/5030265697075200>.
- [82] The Debian Project. Debian. <http://www.debian.org/>.
- [83] The GnuPG Project. The GNU Privacy Guard. <http://www.gnupg.org/>.
- [84] The OpenSSL Project. OpenSSL. <https://www.openssl.org/>.
- [85] Tor Project. Tor Browser Bundle. <https://www.torproject.org/projects/torbrowser.html.en>.
- [86] Guillermo Rauch. Socket.IO. <http://socket.io/>.
- [87] John Resig and the jQuery Team. jQuery. <http://jquery.com/>.
- [88] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev. Using AVL trees for fault-tolerant group key management. *International Journal of Information Security*, 1(2):84–99, 2002.
- [89] Alex Russell. DOM Promises IDL/polyfill. <https://github.com/slightlyoff/Promises>.
- [90] Amit Sahai and Brent Waters. Fuzzy Identity-Based Encryption. 2005.
- [91] Augustin P. Sarr, Philippe Elbaz-Vincent, and Jean-Claude Bajard. A secure and efficient authenticated Diffie-Hellman protocol. In *Proceedings of the 6th European conference on Public key infrastructures, services and applications*, EuroPKI’09, pages 83–98, Berlin, Heidelberg, 2010. Springer-Verlag.
- [92] Roman Schlegel and Duncan S. Wong. Private Friends on a Social Networking Site Operated by an Overly Curious SNP. In *NSS*, pages 430–444, 2012.

- [93] Amre Shakimov, Harold Lim, Ramón Cáceres, On P. Cox, Kevin Li, Dongtao Liu, and Er Varshavsky. Vis-à-vis: Privacy-preserving online social networking via virtual individual servers. In *In COMSNETS*, 2011.
- [94] A. T. Sherman and D. A. McGrew. Key establishment in large dynamic groups using one-way function trees. *Software Engineering, IEEE Transactions on*, 29(5):444–458, 2003.
- [95] Richard I. Shrager. Octave: leasqr function. <http://octave.sourceforge.net/optim/function/leasqr.html>.
- [96] Jack Snoeyink, Subhash Suri, and George Varghese. A Lower Bound for Multicast Key Distribution. In *In Proceedings of IEEE INFOCOM 2001*, volume 47, pages 422–431, New York, NY, USA, feb 2001. Elsevier North-Holland, Inc.
- [97] D. H. Tran, Hai-Long Nguyen, Wei Zhao, and Wee Keong Ng. Towards security in sharing data on cloud-based social networks. In *Information, Communications and Signal Processing (ICICS) 2011 8th International Conference on*, pages 1–5, 2011.
- [98] "uupa". msgpack-javascript. <https://github.com/msgpack/msgpack-javascript>.
- [99] Carlo von Lynx. 13 reasons not to start using PGP, 2008.
- [100] D. Wallner, E. Harder, and R. Agee. Key Management for Multicast: Issues and Architectures. 1999.
- [101] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style Host Authentication with Multi-path Probing. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 321–334, Berkeley, CA, USA, 2008. USENIX Association.
- [102] Brian White. base91.js. <https://github.com/mscdex/base91.js>.
- [103] Alma Whitten and J. D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*, SSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

BIBLIOGRAPHY

- [104] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Trans. Netw.*, 8(1):16–30, feb 2000.
- [105] Xuxin Xu, Lingyu Wang, Amr Youssef, and Bo Zhu. Preventing Collusion Attacks on the One-Way Function Tree (OFT) Scheme. In Jonathan Katz and Moti Yung, editors, *Applied Cryptography and Network Security*, volume 4521, chapter Lecture Notes in Computer Science, pages 177–193. Springer Berlin Heidelberg, 2007.
- [106] Cai-Nicolas Ziegler and Georg Lausen. Spreading Activation Models for Trust Propagation. In *In Proceedings of the IEEE International Conference on e-Technology, e-Commerce, and e-Service*. IEEE Computer Society Press, 2004.
- [107] Phil Zimmerman. Why I Wrote PGP. <http://www.philzimmermann.com/EN/essays/WhyIWrotePGP.html>, 1991.

Appendix A

Schemes

A.1 Database

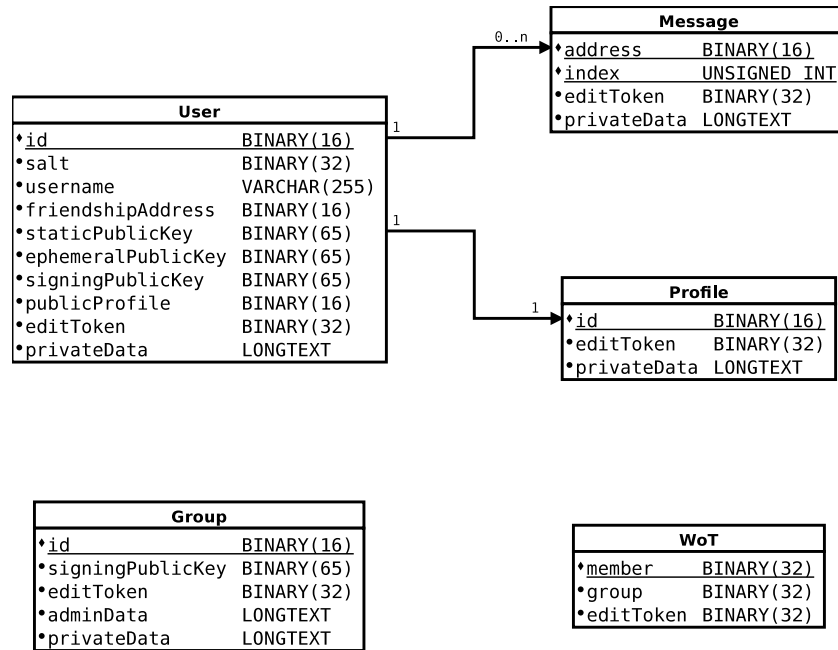


Figure A.1: Scheme of the database tables. `BINARY(x)` indicates a byte-array of length x , `VARCHAR(x)` indicates a string of characters of variable length where x is the maximum value, `LONGTEXT` indicates a variable-length string up to 4 GiB and `UNSIGNED INT` indicates a 32-bit unsigned integer number,

